

Modeling and Efficient Verification of Wireless Ad hoc Networks

Behnaz Yousefi, Fatemeh Ghassemi and Ramtin Khosravi

School of Electrical and Computer Engineering, University of Tehran, Iran
{b.yousefi, fghassemi, r.khosravi}@ut.ac.ir

Abstract. Wireless ad hoc networks, in particular mobile ad hoc networks (MANETs), are growing very fast as they make communication easier and more available. However, their protocols tend to be difficult to design due to topology dependent behavior of wireless communication, and their distributed and adaptive operations to topology dynamism. Therefore, it is desirable to have them modeled and verified using formal methods. In this paper, we present an actor-based modeling language with the aim to model MANETs. We address main challenges of modeling wireless ad hoc networks such as local broadcast, underlying topology, and its changes, and discuss how they can be efficiently modeled at the semantic level to make their verification amenable. The new framework abstracts the data link layer services by providing asynchronous (local) broadcast and unicast communication, while message delivery is in order and is guaranteed for connected receivers. We illustrate the applicability of our framework through two routing protocols, namely flooding and AODVv2-11, and show how efficiently their state spaces can be reduced by the proposed techniques. Furthermore, we demonstrate a loop formation scenario in AODV, found by our analysis tool.

Keywords: state space reduction, mobile ad hoc network, ad hoc routing protocol, Rebeca, actor-based language, model checking

1. Introduction

Applicability of wireless communications is rapidly growing from home networks to satellite transmissions due to their high accessibility and low cost. Wireless communication has a broadcasting nature, as messages sent by each node can be received by all nodes in its transmission range, called *local broadcast*. Therefore, by paying the cost of one transmission, several nodes may receive the message, which leads to lower energy consumption for the sender and throughput improvement [11].

Mobile ad hoc networks (MANETs) consist of several portable hosts with no pre existing infrastructure, such as routers in wired networks or access points in managed (infrastructure) wireless networks. In such networks, nodes can freely change their locations so the network topology is constantly changing. For unicasting a message to a specific node beyond the transmission range of a node, it is needed to relay the message by some intermediate nodes to reach the desired destination. Due to lack of any pre-designed infrastructure and global network topology information, network functions such as routing protocols are devised in a completely distributed manner and adaptive to topology changes. Topology dependent behavior of wireless communication, distributed and adaptation requirements make the design of MANET protocols complicated

Correspondence and offprint requests to: Christiane Notarmarco, Springer-Verlag London Limited, Sweetapple House, Catteshall Road, Godalming, Surrey GU7 3DJ, UK. e-mail: chris@svl.co.uk

and more in need of modeling and verification so that it can be trusted. For instance, MANET protocols like the Ad hoc On Demand distance Vector (AODV) routing protocol [41] has been evolved as new failure scenarios were experienced or errors were found in the protocol design [8, 38, 18].

The actor model [4, 26] has been introduced for the purpose of modeling concurrent and distributed applications. It is an agent-based language introduced by Hewitt [26], extended by Agha to an object-based concurrent computation model [4]. An actor model consists of a set of actors communicating with each other through unicasting asynchronous messages. Each computation unit, modeled by an actor, has a unique address and mailbox; Messages sent to an actor are stored in its mailbox. Each actor is defined through a set of message handlers, called *message servers*, to specify the actor behavior upon processing of each message. In this model, message delivery is guaranteed but is not in-order. This policy implicitly abstracts from effects of the network i.e., node crashes, delays over different routing paths, message conflicts, etc., and consequently makes it a suitable modeling framework for concurrent and distributed applications. Rebeca [52] is an actor-based modeling language which aims to bridge the gap between formal verification techniques and the real world software engineering of concurrent and distributed applications. It provides an operational interpretation of the actor model through a Java-like syntax, which makes it easy to learn and use. Rebeca is supported by a robust model checking tool, named Afra [3], which takes advantage of various reduction techniques [27, 47] to make efficient verification possible. With the aim of reducing the state space, computations, i.e., executions of message servers in actors, are assumed to be instantaneous while message delivery is in-order. Consequently, instructions of message servers are not interleaved and hence, execution of message servers becomes atomic in semantic model and each actor mailbox is modeled through a FIFO queue.

In [56] we introduced bRebeca as an extension to Rebeca, to support broadcast communication which abstracts the *global broadcast communications* [7]. To abstract the effect of network, the order of receipts for two consequent broadcast communications is not necessarily the same as their corresponding sends in an actor model. Hence, each actor mailbox was modeled by a bag. The resulting framework is suitable for modeling and efficient verification of broadcasting protocols above the network layer, but not appropriate for modeling MANETs in two ways: firstly the topology is not defined, and every actor (node) can receive all messages, in other words all nodes are connected to each other. Secondly, as there is no topology defined, mobility is not concerned.

In this paper, we extend the actor-based modeling language bRebeca [56] to address local broadcast, topology, and its changes. The aim of the current paper is to provide a framework to detect malfunctions of a MANET protocol caused by conceptual mistakes in the protocol design, rather than by an unreliable communication. Therefore, the new framework abstracts the data link layer services by providing asynchronous reliable local broadcast, multicast, and unicast communications [40, 49]. Since only one-hop communications are considered, the message delivery is in-order and is guaranteed for connected receivers. Consequently, each actor mailbox is modeled through a queue. The reliable communication services of the data link layer provide feedback (to its upper layer applications) in case of (un)successful delivery. Therefore, our framework provides *conditional unicast* to model protocol behaviors in each scenario (in the semantic model, the status of underlying topology defines the behavior of actors).

The resulting framework provides a suitable means to model the behavior of ad hoc networks in a compositional way without the need to consider asynchronous communications handled by message storages in the computation model. However, to minimize the effect of message storages on the growth of the state space, we exploit techniques to reduce it. Since nodes can communicate through broadcast and a limited form of multicast/unicast, it is possible to consider actors with the same neighbors, topological situations, and local states identical with respect to the counter abstraction technique [5]. The reduced semantics is strong bisimilar to the original one [5].

To examine resistance/adaptation of MANET protocols to changes of the underlying topology, we address mobility via arbitrary changes of the topology at the semantic level. Since network protocols have no control over movement of MANET nodes and mobility is an intrinsic characteristic of such nodes, topology should be implicitly manipulated at the semantics. In other words, with the aim of verifying behaviors of MANET protocols for any mobility scenario, the underlying topology is arbitrarily changed at each semantic state. We provide mechanisms to restrict the random changes in the topology through specifying constraints over the topology. However, this random changes make the state space grow exponentially while the proposed counting abstraction technique becomes invalid. To this end, each state is instead explored for each possible topology and meanwhile topology information is removed from the state. Therefore, two next states only different in their topologies are consolidated together and hence, the state space is reduced considerably.

We establish that the reduced semantics is branching bisimilar to the original one, and consequently a set of properties such as ACTL-X [12] are preserved. The proposed reduction techniques makes our framework scalable to verify some important properties of MANET protocol, e.g., loop freedom, in presence of mobility in a unified model (cf. generating a model for each mobility scenario).

The contributions of this paper can be summarized as follows:

- We extend the computation model of actor model, in particular Rebeca, with the concepts of MANETs, i.e., asynchronous reliable local broadcast/multicast/unicast, topology, and topology changes;
- We apply the counting abstraction in presence of topology as a part of semantic states to reduce state space substantially: actors with the same neighbors, topological situations, and local states are counted together in the counting abstraction technique;
- We show that the soundness of counting abstraction techniques is not preserved in presence of mobility, and propose another technique to reduce the state space.
- We provide a tool that supports both reduction techniques and examines invariant properties automatically. We illustrate scalability of our approach through specification and verification of two MANET protocols, namely flooding and AODV.
- We present a complete and accurate model of the core functionalities of a recent version of AODVv2 protocol (version 11), and investigate its loop freedom property. We detect scenarios over which the property is violated due to maintaining multiple unconfirmed next hops for a route without checking them to be loop free. Furthermore, we verify the monotonic increase of sequence numbers and packet delivery properties using existing model checkers.

Our framework can also be applied to Wireless Mesh Networks (WMNs). Unlike MANETs, WSNs have a backbone of dedicated mesh routers along with mesh clients. Hence, they provide flexibility in terms of mobility: in contrast to MANETs, the clients mobility has limited effect on the overall network configuration as the mesh routers are fixed [32].

The paper is structured as follows. Section 2 briefly introduces bRebeca, provides an overview on the counter abstraction and symmetry reduction techniques, and explains equivalence relations that validate our reduction techniques. Section 3 addresses the main modeling challenges of wireless networks. Section 4 presents our extension to bRebeca for modeling MANETs. In Section 5, we generate the state space compactly with the aim of efficient model checking. To illustrate the applicability of our approach, we specify the core functionalities of AODVv2-11 in Section 6. Then, in Section 7, we discuss the efficiency of our state space generation over two cases studies: the AODV and the flooding-based routing protocol. We illustrate our tool and possible analysis over the models through verification of AODV. Finally, we review some related work in Section 8 before concluding the paper.

2. Preliminaries

2.1. bRebeca

Rebeca [52] is an actor-based modeling language proposed for modeling and verification of concurrent and distributed systems. It has a Java-like syntax familiar to software developers and it is also supported by a tool via an integrated modeling and verification environment [3]. Due to its design principle it is possible to extend the core language based on the desired domain [51]. For example, different extensions have been introduced in various domains such as probabilistic systems [53], real-time systems [45], software product lines [46], and broadcasting environment [56]. As in this paper we intend to extend bRebeca, we briefly review its syntax and semantics.

In bRebeca as the same in Rebeca, actors are the computation units of the system, called rebecs (short for reactive objects), which are instantiated of the defined *reactive classes* in the model.

Rebecs communicate with each other only through broadcasting messages which is fair and asynchronous. It is fair as every sent message eventually will be received and processed by its *potential* receivers. In Rebeca, the rebecs defined as the *known rebecs* of a sender, the sender itself using the “self” keyword, or the sender of current processing message using the keyword “sender” are considered as the potential receivers. However, in bRebeca, it is assumed the network is fully connected and therefore, all rebecs of a model constitute the potential receivers. In other words, a broadcast message is received by all the nodes to which a sender has a

```

1  reactiveclass MNode
2  {
3      statevars
4      {
5          int my_i;
6          boolean done;
7      }
9      msgsrvv initial (int j, boolean starter)
10     {
11         my_i = j;
12         if (starter) {
13             done = true;
14             send(my_i);
15         } else
16             done = false;
17     }
19     msgsrvv send(int i)
20     {
21         if (i < my_i) {
22             if (!done) {
23                 done = true;
24                 send(my_i);
25             }
26         } else {
27             my_i = i;
28             done = true;
29         }
30     }
31 }
32 main
33 {
34     MNode n1(1,false);
35     MNode n2(2,false);
36     MNode n3(3,true);
37     MNode n4(4,false);
38 }

```

Fig. 1. An example of bRebeca: Max-algorithm with 4 nodes

(one-hop/multi-hop) path. So, unlike Rebeca, there is no need for declaring the known rebecs in the reactive class definition. Due to unpredictability of multi-hop communications, the arrival order of messages must be considered arbitrary. Therefore, as the second difference with Rebeca, received messages are stored in an unordered *bag* in each node.

Every reactive class has two major parts, first the *state variables* to maintain the state of the rebec, and second the *message servers* to indicate the reactions of the rebec toward received messages. The local state of a rebec is defined in terms of its state variables together with its message bag. Whenever a rebec receives a message which has no corresponding message server to respond, it simply discards the message. Each rebec has at least one message server called “initial”, which acts like a constructor in object-oriented languages and performs the initialization tasks.

A rebec is said to be *enabled* if and only if it has at least one message in its bag. The computation takes place by removing a message from the bag and executing its corresponding message server atomically, after which the rebec proceeds to process the other messages in its bag (if exists any). Processing a message may have the following consequences:

- it may modify the value of the state variables of the executing rebec, or
- some messages may be broadcast to other rebecs.

Each bRebeca model consists of two substantial parts, the *reactive classes* part and the *main* part. In the *main* part the instances of the reactive classes are created initially while their local variables are initialized.

As an example, Fig. 1 illustrates a simple max finding algorithm modeled in bRebeca, referred to as “Max-Algorithm” [31]. Every node in a network contains an integer value and they intend to find the maximum value of all nodes in a distributed manner. The **initial** message server has a parameter, named **starter**. The rebec with the **starter** value of *true* initiates the algorithm by broadcasting the first message. Whenever a node receives a value from others, compares it with its current value and one of the following scenarios happens:

- if it has not broadcast its value yet and its value is greater than the received one, it would broadcast its value to others;
- if its current value is less than the received one, it would give up broadcasting its value and update its current value to the received one;
- if it has already sent its value, it would only check whether it must updates its value.

This protocol does not work on MANETs as nodes give up to rebroadcast their value after their first broadcast. However, if a node moves and connects to new nodes, it has to re-broadcast its value as its value may be the maximum value in the currently connected nodes. In other words, the Max-Algorithm should find the maximum value among the connected nodes.

2.2. Counter Abstraction

Since model checking is the main approach of verification in Rebeca, we need to overcome the state space explosion, where the state space of a system grows exponentially as the number of components in the system increases. One way to tackle this well-known problem is through applying reduction techniques such as symmetry reduction [10] and counter abstraction [16]. Counter abstraction is indeed a form of symmetry reduction and in case of full symmetry can be used to avoid *constructive orbit problem*, finding a unique representative of each state which is NP-Hard [10], by using *generic representative* as suggested by Emerson in [16]. Nevertheless, the term of *counter abstraction* was first presented in [43] for verification of parameterized systems and further used in different studies such as [5, 28].

The idea of counter abstraction is to record the global state of a system as a vector of counters, one per each local state. Each counter denotes the number of components currently residing in the corresponding local state. In our work, by “components” we mean the actors of the system. This technique turns a model with an exponential size in n , i.e. m^n , into one of a size polynomial in n , i.e. $\binom{n+m-1}{m}$, where n and m denote the number of components and local states, respectively. Two global states S and S' are considered identical up to permutation if for every local state s , the number of components residing in s be the same in the two states S and S' , as permutation only changes the order of elements. For example, consider a system which consists of three components and each has only one variable v_i of boolean type. Three global states $(true, true, false)$, $(false, true, true)$, and $(true, false, true)$ are equivalent and can be abstracted into one global state represented as $(true : 2, false : 1)$.

2.3. Semantic Equivalence

Strong bisimilarity [42] is used as a verification tool to validate the counting abstraction reduction technique on labeled transition systems. A labeled transition system (LTS), is defined by the quadruple $\langle S, \rightarrow, L, s_0 \rangle$ where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a set of transitions, L a set of labels, and s_0 the initial state. Let $s \xrightarrow{\alpha} t$ denote $(s, \alpha, t) \in \rightarrow$.

Definition 2.1 (Strong Bisimilarity). A binary relation $\mathcal{R} \subseteq S \times S$ is called a strong bisimulation if and only if, for any s_1, s'_1, s_2 , and s'_2 and $\alpha \in L$, the following transfer conditions hold:

- $s_1 \mathcal{R} s_2 \wedge s_1 \xrightarrow{\alpha} s'_1 \Rightarrow (\exists s'_2 \in S : s_2 \xrightarrow{\alpha} s'_2 \wedge s'_1 \mathcal{R} s'_2),$
- $s_1 \mathcal{R} s_2 \wedge s_2 \xrightarrow{\alpha} s'_2 \Rightarrow (\exists s'_1 \in S : s_1 \xrightarrow{\alpha} s'_1 \wedge s'_1 \mathcal{R} s'_2).$

Two states s and t are called strong bisimilar, denoted by $s \sim t$, if and only if there exists a strong bisimulation relating s and t .

As explained in Section 1, mobility is addressed through random changes of underlying topology at each semantic state, modeled by un-observable τ -transitions. We propose to remove such transitions while the behavior of each semantic state is explored for all possible topologies. We exploit branching bisimilarity [22] to establish the reduced semantic is branching bisimilar to the original one. Let $\xrightarrow{\tau^*}$ be reflexive and transitive closure of τ -transitions:

- $t \xrightarrow{\tau^*} t;$
- $t \xrightarrow{\tau^*} s$, and $s \xrightarrow{\tau} r$, then $t \xrightarrow{\tau^*} r.$

Definition 2.2 (Branching Bisimilarity). A binary relation $\mathcal{R} \subseteq S \times S$ is called a branching bisimulation if and only if, for any s_1, s'_1, s_2 , and s'_2 and $\alpha \in L$, the following transfer conditions hold:

- $s_1 \mathcal{R} s_2 \wedge s_1 \xrightarrow{\alpha} s'_1 \Rightarrow ((\alpha = \tau \wedge s'_1 \mathcal{R} s_2) \vee (\exists s'_2, s''_2 \in S : s_2 \xrightarrow{\tau^*} s''_2 \xrightarrow{\alpha} s'_2 \wedge s'_1 \mathcal{R} s'_2)),$
- $s_1 \mathcal{R} s_2 \wedge s_2 \xrightarrow{\alpha} s'_2 \Rightarrow ((\alpha = \tau \wedge s_1 \mathcal{R} s'_2) \vee (\exists s'_1, s''_1 \in S : s_1 \xrightarrow{\tau^*} s''_1 \xrightarrow{\alpha} s'_1 \wedge s'_1 \mathcal{R} s'_2)).$

Two states s and t are called branching bisimilar, denoted by $s \simeq_{br} t$, if and only if there exists a branching bisimulation relating s and t .

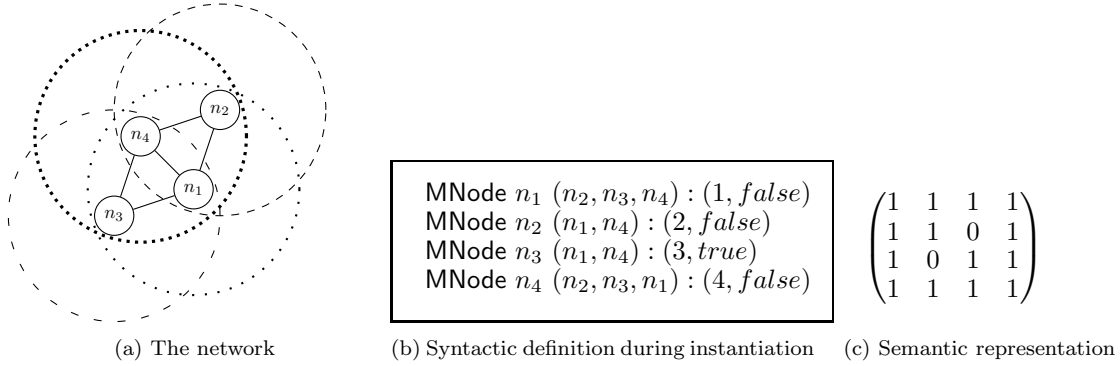


Fig. 2. A sample of initial topology and its corresponding syntactic and semantic representations

3. Modeling Topology and Mobility

In this section, we discuss issues brought up by extending bRebeca to model and verify MANETs, and our solutions to overcome these challenges. We assume that the number of nodes are fixed (to make the state space finite as explained in [15]).

3.1. Network Topology and Mobility

Every rebec represents a node in the MANET models. A node can communicate only with those located in its communication range, so called *connected*. bRebeca does not define “topology” concept as the network graph is considered to be connected, all nodes are globally connected.

Mobility is the intrinsic characteristic of MANET nodes. Furthermore, network protocols have no control over the movement of MANET nodes, and hence, topology changes cannot be specified as a part of the specification. Additionally, to verify a protocol with respect to any mobility scenario, we need to consider all possible topology changes while constructing the state space. To this end, we consider the topology as a part of the states and randomly change the underlying topology at the semantic level. To this aim, a topology is modeled as a $n \times n$ matrix in each (global) state of the semantic model, where n is the number of nodes in the network. Each element of this matrix, denoted by $e_{i,j}$, indicates whether $node_i$ is *connected* to $node_j$ ($e_{i,j} = 1$) or not ($e_{i,j} = 0$). As the transmission ranges of all nodes are assumed to be equal, connectivity is a bidirectional concept, and hence, the resulting matrix will be symmetric. The main diagonal elements are always 1 to make it possible for nodes to unicast messages to themselves (However, in the case of broadcast, our semantic rules prevent a node from receiving its own message, see Section 4.2). Changing the topology is considered an invisible action, modeled by τ transitions, which alters the topology matrix. Each τ -transition represents a set of (bidirectional) link setup/break down.

To setup the initial topology of the network, the *known-rebecs* definitions, provided by the Rebeca language, can be extended to address the connectivity of rebecs. Fig. 2a shows the communication range of the nodes in a simple network. To configure the initial topology of this network, *known-rebecs* of each rebec should be defined as shown in Fig. 2b during its instantiation (cf. Fig. 1). The corresponding semantic representation (as a part of the initial state) is shown in Fig. 2c.

The connectivity matrix has $n \times n$ elements, and since the main diagonal elements are always 1, we have $(n \times n) - n$ changeable elements which can be 1 or 0, and consequently $2^{((n \times n) - n)/2}$ possible topologies. For example, in a network of 4 nodes, we have $2^{(16-4)/2} = 2^6$ possible topologies. Considering all these topologies may lead to state space explosion. Hence, we provide a mechanism to limit the possible topologies by applying some *network constraints* to characterize the set of topologies in terms of (dis)connectivity relations to (un)pin a set of the links among the nodes. We use the notations $con(i, j)$ or $!con(i, j)$ to show that two nodes i and j are connected or disconnected, respectively, and $and(C_1, C_2)$ to denote both C_1 and C_2 hold. For example, $!con(n_1, n_2)$ specifies that n_1 (n_2) never gets connected to n_2 (n_1), in other words, n_1 never enters into n_2 ’s transmission range, and vice versa. Therefore a topology γ is called *valid* for the

network constraint \mathcal{C} , denoted as $\gamma \models \mathcal{C}$, if:

$$\begin{aligned} \gamma \models \text{con}(i, j) &\Leftrightarrow \gamma_{i,j} = 1 & \gamma \models \text{and}(\mathcal{C}_1, \mathcal{C}_2) &\Leftrightarrow \gamma \models \mathcal{C}_1 \wedge \gamma \models \mathcal{C}_2 \\ \gamma \models \text{!con}(i, j) &\Leftrightarrow \gamma_{i,j} = 0 & \gamma \models \text{true} & \end{aligned}$$

where $\gamma_{i,j}$ represents the element $e_{i,j}$ of the corresponding semantic model of γ , and *true* characterizes all possible topologies.

If the only valid topology of a network constraint be equal to the initial topology, then the underlying topology will be static. This case can be useful for modeling WMNs with stable mesh routers with no mesh clients.

3.2. Restricted Delivery Guarantee

The nature of communications in the wireless networks is based on broadcast. The aim of the current paper is to provide a framework to detect malfunctions of a MANET protocol caused by conceptual mistakes in the protocol design, rather than by an unreliable communication. Therefore, we consider the wireless communications in our framework, namely local broadcast, multicast, and unicast, to be asynchronous and reliable in order to abstract the data link layer services. In this way, we abstract the issues related to contention management and collision detection following the approach of [30]. This work abstracts the services of data link layer¹ with the aim to design/analyze MANET protocols irrespective to the network radio model that implements them (its effect is captured by three delays functions). It provides reliable local broadcast communication, with timing guarantees on the worst-case amount of time for a message to be delivered to all its recipients, total amount of time the sender receives its acknowledgement, and the amount of time for a receiver to receive some message among those currently being transmitted by its neighbors, expressed by *delay functions*. Therefore, our approach to specify protocols relying on the abstract data link layer simplifies the study of such protocols, and is valid as its real implementation with such reliable services exists [40, 49]. In these implementations, a node can broadcast/multicast/unicast a message successfully only to the nodes within its transmission range. In the case of unicast, if the sender is located in the receiver transmission range, it will be notified, otherwise it assumes that the transmission was unsuccessful so it can react appropriately. Therefore, we extended bRebeca with *conditional unicast* so that it enables the model to react accordingly based on the status of underlying topology (which defines the delivery status in reliable communications).

Since we only consider one-hop communications (in contrast to the broadcast in bRebeca), the assumption about the unpredictability of multi-hop communications (with different delays) is not valid anymore, and message storages in wRebeca are modeled by queues instead of bags.

4. wRebeca: Syntax and Semantics

In this Section, we extend the syntax of bRebeca, introduced in Section 2.1, with conditional unicast and multicast, topology constraint, and known rebecs to setup the initial topology. Next, we provide the semantics of wRebeca models in terms of labeled transition systems.

4.1. Syntax

The grammar of wRebeca is presented in Fig. 3. It consists of two major parts: reactive classes and main. The definition of reactive classes is almost similar to the ones in bRebeca. However, the main part is augmented with the ConstraintPart, where constraints are introduced to reduce all possible topologies in the network. The instances of the declared reactive classes are defined in the main part, before the ConstraintPart, through indicating the name of a reactive class and an arbitrary rebec name along with two set of parentheses

¹ Data link layer (the second layer of Open Systems Interconnection (OSI) model) is responsible for transferring data across the physical link. It consists of two sublayers: Logical Link Control sublayer (LLC) and Media Access Control sublayer (MAC). LLC is mainly responsible for multiplexing packets to their protocol stacks identified by their IPs, while MAC manages accesses to the shared media.

```

Model ::= ReactiveClass+ Main
Main ::= main {RebecDecl+ ConstraintPart }
List(X) ::= ⟨X,⟩*X | ε
RebecDecl ::= C R (List(R)) : (List(V));
ConstraintPart ::= constraint {Constraint}
Constraint ::= ConstrainDec | ! ConstrainDec | and(Constraint , Constraint)
ConstrainDec ::= con(R , R) | true
ReactiveClass ::= reactiveclass C { StateVars MsgServer* }
StateVars ::= statevars { VarDecl* }
MsgServer ::= msgsrv M(List(T V)) { Statement* }
VarDecl ::= T V;
Statement ::= VarDecl | Assign | Conditional | Loop | Broadcast | Multicast | Unicast | break;
Assign ::= V = Expr;
Conditional ::= if (Expr) Block else Block
Block ::= Statement | { Statement* }
Loop ::= while(Expr) Block
Broadcast ::= M(List(Expr));
Multicast ::= multicast ( V , M(List(Expr)));
Unicast ::= unicast ( Rec , M(List(Expr))) succ : Block unsucc : Block
Rec ::= self | V

```

Fig. 3. wRebeca language syntax: Angle brackets ($\langle \rangle$) are used as metaparentheses. Superscript $*$ indicates zero or more times repetition. The symbols C , R , T , M , and V denote the set of classes, rebec names, types, method and variable names, respectively. The symbol $Expr$ denotes an expression, which can be an arithmetic or a boolean expression.

divided by the character $..$. The first couple of parentheses is used to define the neighbors of the rebec in the initial topology. The second couple of parentheses is used to pass values to the initial message server. Rebecs here communicate through broadcast, multicast, and unicast. In the broadcast statement, we simply use the message server name along with its parameters without specifying the receivers of a message. In contrast, while unicasting/multicasting a message, we also need to specify the receiver/receivers of the message. However, there is no delivery guarantee depending on the location of the receiver. In case of unicasting, the sender can react based on the delivery status. Let $unicast(Rec, M(List(Expr)))$ indicate $unicast(Rec, M(List(Expr)))$ succ : {} unsucc : {} when the delivery status has no effect on the rebec behavior.

In addition to communication statements, there are assignment, conditional, and loop statements. The first one is used to assign a value to a variable. The second is used to branch based on the evaluation of an expression: if the expression evaluates to *true*, then the if part, and otherwise the *else* part would be executed. Assume that $if (Expr) Block$ denotes $if (Expr) Block else \{ \}$. Finally, the third is used to execute a set of statements iteratively as long as the loop condition, i.e., the boolean expression $Expr$, holds. Furthermore, *break* can be used to terminate its nearest enclosing loop statement and transfer the control to the next statement. For the sake of readability, we use $for (T x = Expr_1; Expr_2; Expr_3) \{ Statement^* \}$ to denote $T x = Expr_1; while (Expr_2) \{ Statement^* Expr_3 \}$. A variable can be defined in the scope of message servers as a statement similar to programming languages.

A given wRebeca model is called *well-formed*, if no state variable is redefined in the scope of a message servers, no two state variables, message servers or rebec classes have identical names, identifiers of variables, message servers and classes do not clash, and all rebec instance accesses, message communications and variable accesses occur over declared/specified ones while the number and type of actual parameters correctly match to the formal ones in their corresponding message server specifications. Each *break* should occur within a loop statement. Furthermore, the initial topology should satisfy the network constraint and be symmetric,


```

1  reactiveclass Node
2  {
3      statevars
4      {
5          int IP;
6      }
7
8      msgsrv initial (Boolean source,int ip_)
9      {
10         IP=ip_;
11         if (source==true)
12             relay_packet (55,0,3) ;
13     }
14
15     msgsrv relay_packet(int data,int hopNum,int
16         destination)
17     {
18         if (IP==destination)
19             deliver_packet (data);
20         else if (hopNum<3)
21         {
22             relay_packet (data,hopNum,destination);
23         }
24     }
25     msgsrv deliver_packet(int data)
26     {
27     }
28 }
29
31 main
32 {
33     Node node0 (node1):(true,0);
34     Node node1 (node0,node2,node3):(false,1);
35     Node node2 (node1,node3):(false,2);
36     Node node3 (node1,node2):(false,3);
37
38     constraint
39     {
40         and(con(node0,node1),!con(node0,node2))
41     }
42 }

```

Fig. 4. Flooding protocol in a network consisting of four nodes

i.e., if n_1 is the known rebec of n_2 , then n_2 should be the known rebec of n_1 . By default, the network constraint is `true` if no network constraint is defined, and all the nodes are disconnected if no initial topology is defined.

Example: Flooding protocol is one of the earliest methods used for routing in wireless networks. The flooding protocol modeled by wRebeca is presented in Fig. 4. Every node upon receiving a packet checks whether it is the packet's destination. If so, it processes the message, otherwise, broadcasts the message to its neighbors. To reduce the amount of transferred messages, each message contains a counter, called `hopNum`, which shows how many times it has been re-broadcast. If the `hopNum` is more than the specified bound, it would quit re-broadcasting.

4.2. Semantics

The formal semantics of a well-formed wRebeca is expressed as a labeled transition system (LTS). In the following, we formally define the states, transitions, and initial states of the semantic model generated for a given wRebeca specification. To this aim, the given specification is decomposed into its constituent components, i.e., rebec instances, reactive classes, initial topology, and network constraint represented by the wRebeca model \mathcal{M} . The topology is implicitly changed as long as the given network constraint is satisfied. As explained in Section 1, message server executions are atomic and their statements are not interleaved. Intuitively, the global states of a wRebeca model is defined by the local states of its rebecs and the underlying topology. Consequently, a state transition occurs either upon atomic execution of a message server (i.e., when a rebec processes its corresponding message in its queue), or a random change in the topology (modeled through un-observable τ -transitions).

Let V denote the set of variables ranged over by x , and Val denote the set of all possible values for the variables ranged over by e . Furthermore, we assume the set of the types T consist of the interger and boolean data types, i.e., $T = \{int, bool\}$. We consider the default value $0 \in Val$ for the integer and boolean variables since the boolean values `true` and `false` can be modeled by 1 and 0 in the semantics, respectively. The variable assignment in each scope can be modeled by the valuation function $V \rightarrow Val$ ranged over by θ . A variables valuation can be extended as $\theta \cup \{y \mapsto e\}$. To monitor value assignments regarding the scope management, we specify the set of all environments as $Env = Stack(V \rightarrow Val)$, ranged over by v . Let $upd(v, \{y \mapsto e\})$ extend the variable assignments of the current scope, i.e., the top of the stack, by $\{y \mapsto e\}$ if the stack is not empty. Assume $Stack()$ denotes an empty environment. By entering into a scope, the environment v is updated by $push(\theta, v)$ where θ is empty if the scope belongs to a block (which will be extended by the

declarations in the block). Upon exiting from the scope, it is updated by $pop(v)$ which removes the top of stack. Let $eval(expr, v)$ denote the value of expression $expr$ in the context of environment v , and $v[x := e]$ denote the environment identical to v except that x is assigned to e .

Assume $Seq(D)$ denotes the set of all sequences of elements in D ; we use notation $\langle d_1 \dots d_n \rangle$ and ϵ for a non-empty and empty sequence, respectively. Note that the elements in a sequence may be repeated. A FIFO of elements of D can be viewed as a $Seq(D)$. For instance, $\langle 2 \ 3 \ 2 \ 4 \rangle \in Seq(\mathbb{N})$ denotes a FIFO of natural numbers where its head is 2. For the given FIFO $f : Seq(D)$, assume $f \triangleright d$ denotes the sequence obtained by appending d to the end of f , while $d \triangleright f'$ denotes the sequence with the head d and the tail f' .

A wRebeca model is defined through a set of reactive classes, rebec instances, an initial topology, and a network constraint. Let C denote the set of all reactive classes in the model ranged over by c , R the set of rebec instances ranged over by r , and \mathbb{C} the set of network constraints ranged over by \mathcal{C} . Assume Γ is the set of all possible topologies ranged over by γ . Each reactive class c is described by a tuple $c = \langle V_c, M_c \rangle$ where V_c is the set of class state variables, and M_c the set of message types ranged over by m that its instances can respond to. We assume that for each class c , we have the state variable $self \in V_c$, and $c \in M_c$ which can be seen as its constructor in object-oriented languages. For the sake of simplicity, we assume that messages are parameterized with one argument, so Msg_c , where $M_c = Val \rightarrow Msg_c$, defines the set of all messages that rebec instances of the reactive class c can respond to. The formal parameter of a message can be accessed by $fm : M_c \rightarrow V$. Let *Statement* denote the set of statements ranged over by σ, δ (we use σ^*, δ^* to denote a sequence of statements), and $body : M_c \rightarrow Seq(Statement)$ specify the sequence of statements executed by a message server. A block, denoted by β , is either defined by a statement or a sequence of statements surrounded by braces.

A rebec instance r is specified by the tuple $\langle c, e_0 \rangle$ where $c \in C$ is its reactive class, and e_0 defines the value passed to the message c which is initially put in the rebec's queue. We assume a unique identifier is assigned to each rebec instance. Let $I = \{1 \dots n\}$ denote a finite set of all rebec identifiers ranged over by i and j . Furthermore, we use r_i to denote the rebec instance r with the assigned identifier i . As explained in Section 2.1, a rebec in wRebeca, like Rebeca, holds its received messages in a FIFO (unlike bRebeca, in which messages are maintained in a bag).

All rebecs of the model form a closed model, denoted by $\mathcal{M} = \langle \parallel_{i \in I} r_i, C, \gamma_0, \mathcal{C} \rangle$, where $r_i = \langle c, e_0^i \rangle$ for some $c \in C$ and $\mathcal{C} \in \mathbb{C}$. By default, $\mathcal{C} = true$ and $\forall i, j \leq n (\gamma_{0,i,i} = 1 \wedge (i \neq j \Rightarrow \gamma_{0,i,j} = 0))$ if no network constraint and initial topology were defined. The (global) state of the \mathcal{M} is defined in terms of rebec's local states and the underlying topology.

Definition 4.1. The semantics of the given model $\mathcal{M} = \langle \parallel_{i \in I} r_i, C, \gamma_0, \mathcal{C} \rangle$ is expressed by the labeled transition $\langle S, L, \rightarrow, s_0 \rangle$ where

- $S \subseteq S_1 \times \dots \times S_n \times \Gamma$ is the set of global states such that $(s_1, \dots, s_n, \gamma) \in S$ iff $\gamma \models \mathcal{C}$, and $S_i = Env \times FIFO_i$ is the set of local states of rebec $r_i = \langle c, e_0^i \rangle$ where $FIFO_i = Seq(Msg_c)$ models a FIFO of messages sent to the rebec r_i . Therefore, each s_i can be denoted by the pair (ν_i, f_i) . We use the dot notations $s_i.\nu$ and $s_i.f$ to access the environment and FIFO of rebec i , respectively.
- $L = Act \cup \{\tau\}$ is the set of labels, where $Act = \bigcup_{c \in C} Msg_c$;
- The transition relation $\rightarrow \subseteq S \times L \times S$ is the least relation satisfying the semantic rules in Table 1;
- s_0 is the initial state, and is defined by the combination of initial states of rebecs and the initial topology, i.e., $s_0 = (s_0^1, \dots, s_0^n, \gamma_0)$, where for the rebec $r_i = \langle c, e_0^i \rangle$, $s_0^i = (push(\theta_0, stack()), \langle c(e_0^i) \rangle)$ which denotes that the class variables (i.e., V_c) are initialized to the default value, denoted by θ_0 , and its queue includes only the message $c(e_0^i)$, and $\gamma_0 \models \mathcal{C}$.

To describe the semantics of transitions in wRebeca in Table 1, we exploit an auxiliary transition relation $\rightsquigarrow_{\gamma} \subseteq Env \times FIFO_1 \times \dots \times FIFO_n \times Seq(Statement) \rightarrow Env \times FIFO_1 \times \dots \times FIFO_n \times \{\top, \perp\}$ to address the effect of statement executions on the given environment of the rebec (which executes the statements) and the queue of all rebecs. Upon execution, the statements are either successfully terminated, denoted by \top , or abnormally terminated, denoted by \perp . Let ζ range over $\{\top, \perp\}$. Rule *Term* explains that an empty statement terminates successfully. The effect of an assignment statement, i.e., $x := expr$, is that the value of variable x is updated by $eval(expr, \nu_i)$ in ν_i as explained by rule *Ass*. The variable declaration $T \ x$; extends the variable valuation corresponding to the current scope by the value assignment $x \mapsto 0$, where 0 is the default value for the types of T , as explained in rule *VDecl*. The behavior of a block is expressed by the rule *Block*, based on the behavior of the statements (in its scope) on the environment $push(\emptyset, \nu_i)$,

Table 1. wRebeca natural semantic rules

<i>Term</i> :	$\nu_i, f_1, \dots, f_n, \epsilon \rightsquigarrow_\gamma \nu_i, f_1, \dots, f_n, \top$
<i>Ass</i> :	$\nu_i, f_1, \dots, f_n, x := \text{expr}; \rightsquigarrow_\gamma \nu_i[x := \text{eval}(\text{expr}, \nu_i)], f_1, \dots, f_n, \top$
<i>VDecl</i> :	$\nu_i, f_1, \dots, f_n, T \ x; \rightsquigarrow_\gamma \text{upd}(\nu_i, \{x \mapsto 0\}), f_1, \dots, f_n, \top$
<i>Block</i> :	$\frac{\text{push}(\emptyset, \nu_i), f_1, \dots, f_n, \sigma^* \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}{\nu_i, f_1, \dots, f_n, \{\sigma^*\} \rightsquigarrow_\gamma \text{pop}(\nu'_i), f'_1, \dots, f'_n, \zeta}$
<i>Cond₁</i> :	$\frac{\text{eval}(\text{expr}, \nu_i) = \text{true} \quad \nu_i, f_1, \dots, f_n, \beta_1 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}{\nu_i, f_1, \dots, f_n, \text{if } \text{expr } \beta_1 \text{ else } \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}$
<i>Cond₂</i> :	$\frac{\text{eval}(\text{expr}, \nu_i) = \text{false} \quad \nu_i, f_1, \dots, f_n, \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}{\nu_i, f_1, \dots, f_n, \text{if } \text{expr } \beta_1 \text{ else } \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}$
<i>Loop₁</i> :	$\frac{\begin{array}{c} \text{eval}(\text{expr}, \nu_i) = \text{true} \\ \nu_i, f_1, \dots, f_n, \beta \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \top \\ \nu'_i, f'_1, \dots, f'_n, \text{while}(\text{expr}) \beta \rightsquigarrow_\gamma \nu''_i, f''_1, \dots, f''_n, \top \end{array}}{\nu_i, f_1, \dots, f_n, \text{while}(\text{expr}) \beta \rightsquigarrow_\gamma \nu''_i, f''_1, \dots, f''_n, \top}$
<i>Loop₂</i> :	$\frac{\begin{array}{c} \text{eval}(\text{expr}, \nu_i) = \text{true} \\ \nu_i, f_1, \dots, f_n, \beta \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \perp \end{array}}{\nu_i, f_1, \dots, f_n, \text{while}(\text{expr}) \beta \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \top}$
<i>Loop₃</i> :	$\frac{\text{eval}(\text{expr}, \nu_i) = \text{false}}{\nu_i, f_1, \dots, f_n, \text{while}(\text{expr}) \beta \rightsquigarrow_\gamma \nu_i, f_1, \dots, f_n, \top}$
<i>BCast</i> :	$\nu_i, f_1, \dots, f_n, m(\text{expr}); \rightsquigarrow_\gamma \nu_i, f'_1, \dots, f'_n, \top$, where $\forall k \leq n(k \neq i \wedge (\gamma_{i,k} == 1) \Rightarrow [f'_k = f_k \triangleright m(\text{eval}(\text{expr}, \nu_i))][f'_k = f_k])$
<i>MCast</i> :	$\nu_i, f_1, \dots, f_n, \text{multicast}(\text{rcvs}, \text{expr}); \rightsquigarrow_\gamma \nu_i, f'_1, \dots, f'_n, \top$, where $\forall k \leq n(k \in \text{rcvs} \wedge (\gamma_{i,k} == 1) \Rightarrow [f'_k = m(\text{eval}(\text{expr}, \nu_i)) \triangleright f_k][f'_k = f_k])$
<i>UCast₁</i> :	$\frac{\begin{array}{c} (\gamma_{i,j} == 1) \\ f'_j = f_j \triangleright m(\text{eval}(\text{expr}, \nu_i)) \wedge \forall k \neq j(f'_k = f_k) \\ \nu_i, f'_1, \dots, f'_n, \beta_1 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta \end{array}}{\nu_i, f_1, \dots, f_n, \text{unicast}(j, m(\text{expr})) \text{ succ} : \beta_1 \text{ unsucc} : \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}$
<i>UCast₂</i> :	$\frac{(\gamma_{i,j} == 0) \quad \nu_i, f_1, \dots, f_n, \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}{\nu_i, f_1, \dots, f_n, \text{unicast}(j, m(\text{expr})) \text{ succ} : \beta_1 \text{ unsucc} : \beta_2 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \zeta}$
<i>Seq₁</i> :	$\frac{\nu_i, f_1, \dots, f_n, \sigma_1 \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \top \quad \nu'_i, f'_1, \dots, f'_n, \sigma_2^* \rightsquigarrow_\gamma \nu''_i, f''_1, \dots, f''_n, \zeta}{\nu_i, f_1, \dots, f_n, \sigma_1 \sigma_2^* \rightsquigarrow_\gamma \nu''_i, f''_1, \dots, f''_n, \zeta}$
<i>Seq₂</i> :	$\nu_i, f_1, \dots, f_n, \text{break}; \sigma^* \rightsquigarrow_\gamma \nu_i, f_1, \dots, f_n, \perp$
<i>Handle</i> :	$\frac{\begin{array}{c} s_i.f = m(e) \triangleright f_i \wedge \forall k \neq i(f_k = s_k.f) \\ \nu_i = \text{push}(\{fm(m) \mapsto e\}, s_i.\nu) \\ \nu_i, f_1, \dots, f_n, \text{body}(m) \rightsquigarrow_\gamma \nu'_i, f'_1, \dots, f'_n, \top \end{array}}{(s_1, \dots, s_n, \gamma) \xrightarrow{m(e)} (s'_1, \dots, s'_n, \gamma)}, \text{ where } \forall k \neq i(s'_k = (s_k.\nu, f'_k)) \wedge s'_i = (\text{pop}(\nu'_i), f'_i)$
<i>Mov</i>	$(s_1, \dots, s_n, \gamma) \xrightarrow{\tau} (s_1, \dots, s_n, \gamma')$, where $\gamma' \models \mathcal{C}$

where the empty valuation function may be extended by the declarations in the scope (by rule *VDecl*). Thereafter, to find the effect of the block, the last scope is pop off the environment. Rules *Cond*_{1,2} specify the effect of the *if* statement: if $eval(expr, \nu_i)$ evaluates to *true*, its effect is defined by the effect of executing the *if* part, otherwise the *else* part. Rules *Loop*₁₋₃ explain the effect of the *while* statement; If the loop condition evaluates to *true*, the effect of *while* statement is defined in terms of the effect of its body by the rules *Loop*_{1,2}, otherwise it terminates immediately as specified by the rule *Loop*₃. If the body of the *while* statement terminates successfully, the effect of *while* statement is defined in terms of the effect of the *while* statement on the resulting environment and queues of its body execution as explained by *Loop*₁. Rule *Loop*₂ expresses that if the body of the *while* statement terminates abnormally (due to a *break* statement) while its condition evaluates to *true*, then it terminates successfully while taking the effect of its body execution into account. The effect of a sequence of statements is specified by the rules *Seq*_{1,2}. Upon successful execution of a statement, the effect of its next statements is considered (rule *Seq*₁). A *break* statement makes all its next statements be abandoned (rule *Seq*₂).

The post-conditions in the form of $b \Rightarrow [C_1][C_2]$ in the rules *BCast* and *MCast* abbreviates $(b \Rightarrow C_1) \wedge (\neg b \Rightarrow C_2)$. The effect of broadcast and multi-cast communications are specified by the rules *BCast* and *MCast*, respectively: the message $m(eval(expr, \nu_i))$ is appended to the queue of all connected nodes to the sender in case of broadcast, and all connected nodes among the specified receivers (i.e., *rcvs*) in case of multi-cast. Rules *UCast*_{1,2} express the effect of unicast communication upon its delivery status. If the communication was successful (i.e., the sender was connected to the receiver), the message is appended to the queue of the receiver while the effect of *succ* part is also considered (rule *UCast*₁), otherwise only the effect of *unsucc* part is considered (rule *UCast*₂).

Rule *Handle* expresses that the execution of a wRebeca model progresses when a rebec processes the first message of its queue. In this rule, the message $m(e)$ is processed by the rebec r_i as $s_i.f = m(e) \triangleright f_i$. To process this message, its corresponding message server, i.e., $body(m)$ is executed. The effect of its execution is captured by the transition relation \leadsto_γ on the environment of r_i , updated by the variable assignment $\{fm \mapsto e\}$ for the scope of the message server of m , and the queue of all rebecs while message $m(e)$ is removed from the queue of r_i . Finally, rule *Mov* specifies that the underlying topology is implicitly changed at the semantic level, and the new topology satisfies \mathcal{C} .

Example: Consider the global state $(s_0, s_1, s_2, s_3, \gamma)$ such that $s_0 = ((\{IP \mapsto 0\}, \langle relay_packet(55, 0, 3) \rangle), s_1 =$

$(\{IP \mapsto 1\}, \epsilon), s_2 = (\{IP \mapsto 2\}, \epsilon), s_3 = (\{IP \mapsto 3\}, \epsilon)$, and $\gamma : \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$ for the wRebeca model in Fig. 4

where $\{IP \mapsto i\}$ denotes $push(\{IP \mapsto i\}, Stack())$. Regarding our rules, the following transition is derived:

$$\begin{array}{c}
 \frac{\nu_2, \epsilon, \epsilon, \epsilon, \epsilon, hopNum ++ \leadsto_\gamma \nu_3, \epsilon, \epsilon, \epsilon, \epsilon, \top \quad \nu_3, \epsilon, \epsilon, \epsilon, \epsilon, rel(data, \dots) \leadsto_\gamma \nu_3, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top}{\nu_2, \epsilon, \epsilon, \epsilon, \epsilon, hopNum ++; rel(data, \dots) \leadsto_\gamma \nu_3, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top} Seq_1 \\
 \frac{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, \{hopNum ++; rel(data, \dots)\} \leadsto_\gamma \nu_4, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top : (*)}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, \{hopNum ++; rel(data, \dots)\} \leadsto_\gamma \nu_4, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top} Block \\
 \frac{eval(hopNum < 3, \nu_1) = true \quad (*)}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if(hopNum < 3) \dots \leadsto_\gamma \nu_4, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top} Cond_1 \\
 \frac{eval(IP == des, \nu_1) = false \quad \nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if(IP == \dots \leadsto_\gamma \nu'_1, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top)}{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if(IP == \dots \leadsto_\gamma \nu'_1, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top} Cond_2 \\
 \frac{\nu_1, \epsilon, \epsilon, \epsilon, \epsilon, if(IP == \dots \leadsto_\gamma \nu'_1, \epsilon, \langle rel(55, 1, 3) \rangle, \epsilon, \epsilon, \top)}{(s_0, s_1, s_2, s_3, \gamma) \xrightarrow{rel(55, 0, 3)} (s'_0, s'_1, s_2, s_3, \gamma)} Handle
 \end{array}$$

where $\nu_1 = push(\{data \mapsto 55, hopNum \mapsto 0, des \mapsto 3\}, \{IP \mapsto 0\})$, $\nu_2 = push(\emptyset, \nu_1)$, $\nu_3 = \nu_2[hopNum := 1]$, $\nu_4 = pop(\nu_3)$, $\nu'_1 = pop(\nu_4)$, $s'_0 = (\{IP \mapsto 0\}, \epsilon)$, and $s'_1 = (\{IP \mapsto 1\}, \langle rel(55, 1, 3) \rangle)$. Note that *des* denotes **destination**, and *rel* refers to **relay_packet** message. By the rule *Handle*, the message $rel(55, 0, 3)$ in the queue of *node*₀ is processed. To this aim, the body of its message server, i.e., *if* ($IP == \dots$) is executed. Since $eval(IP == des, \nu_1) = false$, by the rule *Cond*₂, the *else* part (i.e., *if* ($hopNum < 3$) ...) is executed. Due to $eval(hopNum < 3, \nu_1) = true$ by the rule *Cond*₁, the *if* part is executed.

5. State Space Reduction

We extend application of counting abstraction technique to wRebeca models when the topology is static. To this end, the local states of rebecs and their neighborhoods are considered. Later, we inspect the soundness of

counting abstraction technique in presence of mobility. As a consequence, we propose a reduction technique based on removal of τ -transitions. Recall that the topology is static when the only valid topology of the network constraint be equal to the initial topology.

5.1. Applying Counter Abstraction

Assume S_c is the set of local states that the instances of the reactive class c can take (i.e., $S_c = Env_c \times FIFO_c$) and I is the set of rebecs identifiers. To apply the counter abstraction, rebecs with an identical local state and neighbors that are *topologically equivalent* are counted together. Two nodes $i, j \in I$ are said to be topologically equivalent, denoted by $i \approx_\gamma j$ iff $\forall k \in I \setminus \{i, j\} (\gamma_{ik} = \gamma_{jk})$. Intuitively, two topologically equivalent nodes have the same neighbors (except themselves). So if either one broadcasts, the same set of nodes (except themselves) will receive, and if they are also connected to each other, its counterpart (that is symmetric to the sender) will receive. Nodes in $\mathcal{N} \subseteq I$ are called topologically equivalent iff $\forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\} (i \approx_\gamma j \wedge \forall k, l \in \mathcal{N} ((k \neq l) \Rightarrow \gamma_{ik} = \gamma_{kl}))$. This definition implies that all topologically equivalent nodes should be either all connected to each other, or disconnected, while they should have the same neighbors (except themselves). Therefore, topologically equivalent nodes will affect the same nodes when either one broadcasts. Hence, topologically equivalent nodes with an identical local state can be aggregated. To this aim, nodes of the underlying topology are partitioned into the maximal sets of topologically equivalent nodes, denoted by $\mathcal{N}_1, \dots, \mathcal{N}_\ell$. We define the set of *distinct local states* as $S^d = \bigcup_{c \in C} S_c$, and the set of topology equivalence classes as $\mathbb{T} = \bigcup_{i=1}^\ell \mathcal{N}_i$. Consequently, each global state $(s_1, \dots, s_n, \gamma)$ is abstracted by a vector of elements $(s_i^d, \mathcal{N}_i) : c_i$ where $s_i^d \in S^d$, $\mathcal{N}_i \in \mathbb{T}$, and c_i is the number of nodes in the topology equivalence class \mathcal{N}_i that reside in the very local state s_i^d . The reduced global state, called *abstract global state*, is presented as follows, where n and m donate the numbers of all rebecs and distinct local states (i.e., $m = |S^d|$), respectively:

$$S = ((s_1^d, \mathcal{N}_1) : c_1, \dots, (s_k^d, \mathcal{N}_k) : c_k), \forall i \leq k (c_i > 0 \wedge \mathcal{N}_i \in \mathbb{T}), \sum_{i=1}^k c_i = n, k \leq m$$

For instance, nodes n_1 , n_4 , and n_2 , n_3 in Fig. 2a have the same neighbors, so if their state variables and queue contents be the same, then they can be counted together.

Recall that when the underlying topology is static, a global state only may change upon processing a message by a rebec, since in wRebeca the body of message servers execute atomically. Thus, its corresponding abstract global state also may change upon processing a message by a rebec.

Counting abstraction is beneficial when the reactive classes do not have a variable that will be assigned uniquely to its instances, such as “unique address” as a state variable (Note that at the semantics, rebecs have identifiers which are not a part of their local states). For example, the counting abstraction is not effective on the specification of the *flooding protocol* given in Fig. 4, since its nodes are identified uniquely by their IPs, and hence their state variables can not be collapsed. Therefore, to take benefits of this abstraction, we revise the example in the way that nodes are not distinguished by their IPs. To this aim, the IP variable is replaced by the boolean variable **destination** which identifies the sink node, while the last parameter of **relay_packet** message server is removed. The revised version is shown in Fig. 5.

The reduction takes place on-the-fly while constructing the state space. To this end, each global state $(s_1, \dots, s_n, \gamma)$ is transformed into the form $((s_1^d, \mathcal{N}_1) : n_1, (s_2^d, \mathcal{N}_2) : n_2, \dots, (s_k^d, \mathcal{N}_k) : n_k)$ such that $n_i \subseteq \mathcal{N}_i$ is the set of node identifiers that are topologically equivalent with the local states equal to s_i^d , where $\mathcal{N}_i \in \mathbb{T}$. This new presentation of the global state is called *transposed global state*. The sets n_i are leveraged to update the states of the potential receivers (known by the underlying topology) when a communication occurs. To generate the abstract global states, each transposed global state is processed by taking an arbitrary node from the set assigned to a distinct local state and a topology equivalence class if the distinct local state constitutes of a non-empty queue. The next transposed global state is computed by executing the message handler of the head message in the queue. This is repeated for all the pairs of a distinct local state and a topology equivalence class of the transposed global state. After generating all the next transposed global states of a transposed state, the transposed state is transformed into its corresponding abstract global state by replacing each n_i by $|n_i|$. A transposed global state is processed only if its corresponding abstract global state has not been previously computed. During state space generation, only the abstract global states are stored. Fig. 6 illustrates a global state and its corresponding transposed global state. It is assumed that the

```

1  reactiveclass Node
2  {
3      statevars
4      {
5          boolean destination;
6      }
7
8      msgsrvv initial (boolean source, boolean dest)
9      {
10         destination=dest;
11         if (source==true)
12             relay_packet (55,1);
13     }
14
15     msgsrvv relay_packet(int data, int hopNum)
16     {
17         if (destination==true)
18             deliver_packet (data);
19         else if (hopNum<2)
20
21         {
22             hopNum++;
23             relay_packet (data, hopNum);
24         }
25     }
26     msgsrvv deliver_packet(int data)
27     {
28
29     }
30 }
31
32 main
33 {
34     Node node0 (node1):(true,false);
35     Node node1 (node0,node2,node3):(false,false);
36     Node node2 (node1,node3):(false,false);
37     Node node3 (node1,node2):(false,true);
38
39 }

```

Fig. 5. The revised version of the flooding protocol to make counter abstraction applicable in the network consisting of four nodes

$$\left(\begin{array}{l} (\{i \mapsto 1\}, \epsilon), \\ (\{i \mapsto 2\}, \langle msg \rangle), \\ (\{i \mapsto 1\}, \epsilon), \\ (\{i \mapsto 0\}, \epsilon), \end{array} \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \right)$$

(a) Before applying counter abstraction

$$\left(\begin{array}{l} (\{i \mapsto 1\}, \epsilon), \{1, 3\} : \{1, 3\}, \\ (\{i \mapsto 2\}, \langle msg \rangle), \{2, 4\} : \{2\}, \\ (\{i \mapsto 0\}, \epsilon), \{2, 4\} : \{4\} \end{array} \right)$$

(b) After applying counter abstraction

Fig. 6. A global state and its corresponding transposed global state: assume $\{i \mapsto e\}$ denotes $push(\{i \mapsto e\}, Stack())$

network consists of four nodes of the reactive class with only one state variable i and message server msg . Each row in Fig. 6a represents a local state, i.e., valuation of the local state variable and message queue, while each row in Fig. 6b represents a distinct local state and a set of topologically equivalent identifiers together with those nodes of the set that reside in that distinct local state. As the topology is static, it can be removed from the abstract/transposed global states. Furthermore, each topology equivalence class of nodes can be represent by its unique representative, e.g., the one with the minimum identifier.

The following theorem states that applying counter abstraction does preserve semantic properties of the model modulo strong bisimilarity. To this aim, we prove that states that are counted together are strong bisimilar. For instance, the global state similar to the one in Fig. 6a except that the distinct local states of nodes 2 and 4 are vice versa, is also mapped into the same abstract global state that corresponds to Fig. 6b.

Theorem 5.1 (Soundness of Counting Abstraction). Assume the two global states S_1 and S_2 such that for all pair of $s^d \in S^d$ and $\mathcal{N} \in \mathbb{T}$, the numbers of topologically equivalent nodes of \mathcal{N} that have the distinct local state s^d are the same in S_1 and S_2 . Then they are strong bisimilar.

Proof. Since the topology is static, the only transitions these states have are the result of processing messages in their rebec queues. Suppose $S_1 \xrightarrow{m(e)} S'_1$, since there is a node i with the local state (ν_i, f_i) in the topology equivalence class \mathcal{N} , where $m(e)$ is the head of f_i using the semantic rule *Handle* in Table 1. Assume that i belongs to the topologically equivalent nodes $\mathcal{N}_1 \subseteq \mathcal{N}$, where $((\nu_i, f_i), \mathcal{N}) : \mathcal{N}_1$ is an element of the transposed global state corresponding to S_1 . Due to the assumption, there exists the topologically equivalent nodes $\mathcal{N}_2 \subseteq \mathcal{N}$ in S_2 with the distinct local state (ν_i, f_i) that $|\mathcal{N}_1| = |\mathcal{N}_2|$. We choose the random node j in \mathcal{N}_2 and prove that it could trigger the same transition as i . We claim that $\forall (s_k^d, \mathcal{N}')$, the number of the nodes in the topology equivalence class \mathcal{N}' that are a neighbor of i , denoted by nb_i , and reside in the local state s_k^d is the same to the number of the nodes in the topology equivalence class \mathcal{N}' that are a neighbor of j , denoted by nb_j , with the local state s_k^d . Assume for the arbitrary transposed global state element (s_l^d, \mathcal{N}')

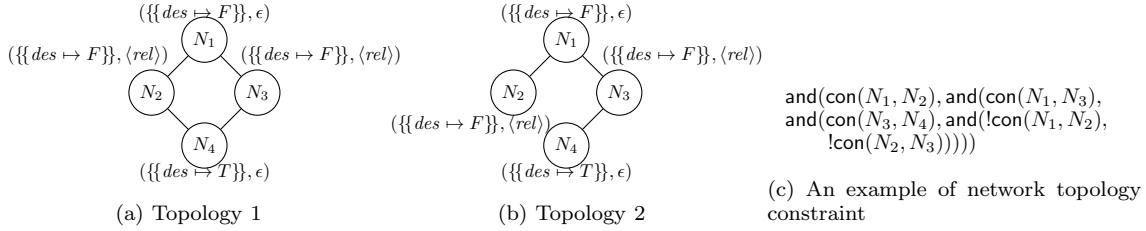


Fig. 7. Two possible topologies for the given constraint on the flooding protocol

this does not hold, and without loss of generality, nb_i has more topologically equivalent nodes than nb_j in (s_i^d, \mathcal{N}'') . As the links are bidirectional, due to the definition of abstract/transposed global states, i is the neighbor of the all nodes in \mathcal{N}'' . Furthermore, as the topology is the same for S_1 and S_2 and $i, j \in \mathcal{N}$, then j is also the neighbor of the all nodes in \mathcal{N}'' . However, due to the assumption, the numbers of topologically equivalent nodes of \mathcal{N}'' in S_1 and S_2 that have the distinct local state s_i^d are the same. So there are some topologically equivalent nodes of \mathcal{N}'' with the local state s_i^d that are not in nb_j which contradicts to the fact that j is the neighbor of the all nodes in \mathcal{N}'' .

As both i and j handle the same message, they execute the same message server, and consequently the effects on their own local state and their neighbors will be the same. Therefore, $S_2 \xrightarrow{m(e)} S'_2$ while $\forall (s_o^d, \mathcal{N}^*)$ the number of topologically equivalent nodes from the equivalence class \mathcal{N}^* in S'_2 that have the distinct local state s_o^d is the same to S'_1 . The same discussion holds when $S_2 \xrightarrow{m(e)} S'_2$. \square

As mentioned before, the reduction is applicable if the network is static. It is due to the fact that nodes' neighborhood may change and the nodes which are considered to be in the same equivalence class in some state their neighbors may be changed in the next state. Consider the flooding protocol (Fig. 5) for the two topologies shown in Fig. 7a and Fig. 7b (satisfying the network constraint in Fig. 7c). By applying the counter abstraction, nodes N_2 and N_3 are considered equivalent under the topology 1, but not under the topology 2.

To illustrate that counter abstraction is not applicable to the system with a dynamic topology, Fig. 8 shows a part of the state space of the flooding protocol with a change in the underlying topology (from Fig. 7a to Fig. 7b) with/without applying counter abstraction, where only these two topologies are possible. As we expected, the reduced state space is not strong bisimilar (see Section 2.3 for the definition) to its original state space. During transposed global state generation, the next state is only generated for the node 2 with the distinct local state $(\{\{des \mapsto F\}\}, \langle rel \rangle)$ from the equivalence class $\{2, 3\}$. Therefore, it is obvious that the next states in the left LTS of Fig. 8 can be matched to the states with the solid borders in the right LTS. However, the solid bordered states are not strong bisimilar to the dotted ones in the right LTS. As explained in Section 1, the reduced LTS should be strong bisimilar to its original one to preserve all properties of its original model.

To take more advantages of the reduction technique, the message storages can be modeled as bags. However, such an abstraction results more interleavings of messages which do not necessarily happen in the reality, and hence, an effort to inspect if a given trace (of the semantic model) is a valid scenario in the reality is needed. This effort is tolerable if the state space reduces substantially.

5.2. Eliminating τ -Transitions

Instead of modifying the underlying topology, modeled by τ -transitions, messages can be processed with respect to all possible topologies (not only to the current underlying topology). Therefore, all τ -transitions are eliminated and only those that correspond to processing of messages are kept. The following Theorem expresses that removal of τ -transitions and topology information from the global states preserves properties of the original model modulo branching bisimulation such as ACTL-X [12]. In fact, by exploiting the result of [12] about the correspondence between the equivalence induced by ACTL-X and branching bisimulation, the ACTL-X fragments of CACTL [21], introduced to specify MANET properties, and μ -calculus are also

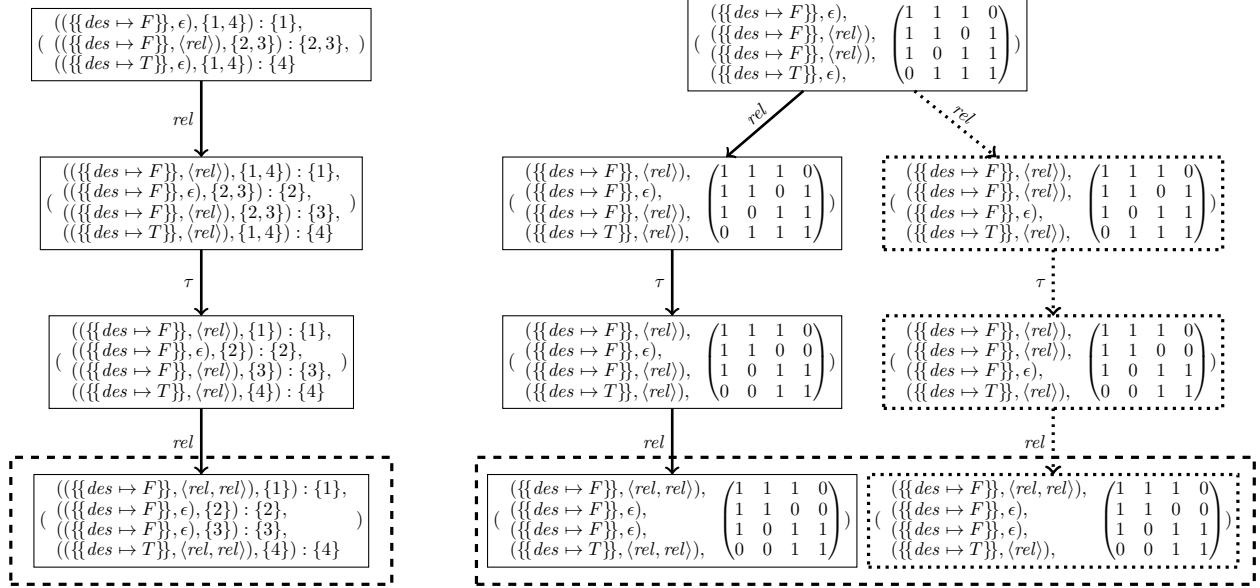


Fig. 8. Comparing a part of flooding protocol state space with/without applying counter abstraction in a dynamic network. Two dashed bordered states are not strong bisimilar since in the right figure there is a global state in which only one node has two *rel* messages in its queue while in the left figure there are two nodes with queues containing two *rel* messages. Note that *T*, *F* stand for *true*, *false*, *des* denotes *destination*, and *rel* refers to *relay_packet* messages, for simplicity the message parameters are not shown in the Figure.

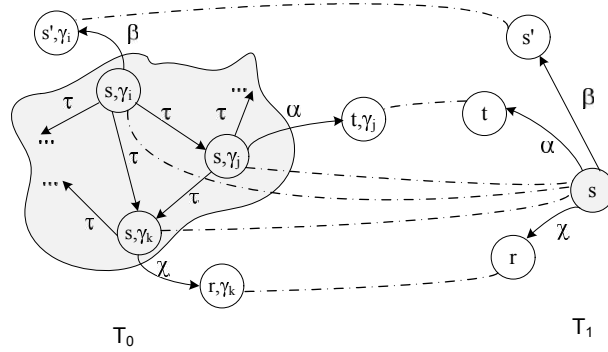


Fig. 9. Relation \mathcal{R} matches states (s, γ) of T_0 to s of T_1 .

preserved. We show in Section 7.3 that important properties of MANET protocols can be still verified over reduced state spaces.

Theorem 5.2 (Soundness of τ -Transition Elimination). For the given LTS $T_0 \equiv \langle S \times \Gamma, \rightarrow, L, (s_0, \gamma_0) \rangle$, assume $(s, \gamma) \xrightarrow{\alpha} (t, \gamma') \Rightarrow (\alpha \neq \tau \wedge \gamma = \gamma') \vee (\alpha = \tau \wedge (\gamma = \gamma' \vee s = t))$, and $\forall \gamma, \gamma' \in \Gamma : (s, \gamma) \xrightarrow{\tau} (s, \gamma')$. Consider $T_1 \equiv \langle S, \rightarrow', L, s_0 \rangle$ such that $(s, \alpha, t) \in \rightarrow'$ if and only if $(s, \gamma) \xrightarrow{\alpha} (t, \gamma') \wedge (\gamma = \gamma')$, then $T_0 \simeq_{br} T_1$.

Proof. Construct $\mathcal{R} = \{((s, \gamma), s) | s \in S, \gamma \in \Gamma\}$ as shown in Figure 9. We show that \mathcal{R} is a branching bisimulation. To this aim, we show that it satisfies the transfer conditions of Definition 2.2. For an arbitrary relation $(s, \gamma) \mathcal{R} s$, assume $(s, \gamma) \xrightarrow{\alpha} (t, \gamma')$. If $\alpha = \tau$, then two cases can be distinguished: (1) either $\gamma \neq \gamma'$, and hence by definition of T_0 , $s = t$ concluding $(t, \gamma') \mathcal{R} s$, (2) or $\gamma = \gamma'$ and by definition of T_1 , $s \xrightarrow{\alpha} t$, and $(t, \gamma') \mathcal{R} t$. Otherwise (i.e., $\alpha \neq \tau$) by definition of T_0 , $\gamma = \gamma'$ and hence by definition of T_1 , $s \xrightarrow{\alpha} t$, and

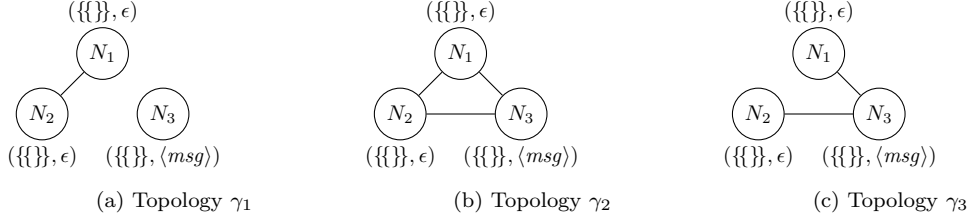


Fig. 10. All possible topologies considered during state space generation of Fig. 11

$(t, \gamma') \mathcal{R} t$. Whenever $s \xrightarrow{\alpha'} t$, then by definition of T_1 there exists γ' such that $(s, \gamma') \xrightarrow{\alpha} (t, \gamma')$ and hence, $(t, \gamma') \mathcal{R} t$. Furthermore by definition of T_0 , $(s, \gamma) \xrightarrow{\tau} (s, \gamma')$ and $(s, \gamma') \mathcal{R} s$. Consequently \mathcal{R} is a branching bisimulation relation. \square

As an example, consider a network which consists of three nodes, which are the instances of a reactive class with no state variable and only one message, *msg*. The message server *msg* has only one statement to broadcast the message *msg* to its neighbors. We assume that the set of all possible topologies are restricted by a network constraint to the three topologies depicted in Fig. 10. Consider the global state that only N_3 has one *msg* in its queue.

The state space of the above imaginary model before reduction is presented in Fig. 11a, where transitions take place by processing messages or changing the topology. Fig. 11b illustrates the state space after eliminating τ -transitions and topology information. As it is obvious, keeping the connectivity information in the global state is trivial, since in each state all possible topologies will be considered. In this approach, transition labels are paired with the topology to denote the topology dependant behavior of transitions. The two transitions labeled with γ_2 and γ_3 can be merged by characterizing the links that make communication from N_3 to N_1 and N_2 ; i.e., from the sender to the receivers. Such links can be characterized by the network constraints depicted in Fig. 11c. In this model, a state is representative of all possible topologies. The resulting semantic model, called *Constrained Labeled Transition System* (CLTS), was introduced in [20] as the semantic model to compactly model MANET protocols. Another advantage of CLTS is its model checker to verify topology-dependant behavior of MANETs [21]. The properties in wireless networks are usually pre-conditioned to existence of a path between two nodes. This model checker takes benefit of network constraints over transitions and assures a property holds if the required paths hold (inferred from the traversed network constraints).

6. Modeling the AODVv2 protocol

To illustrate the applicability of the proposed modeling language, the AODVv2 ²(i.e., version 11) protocol is modeled. The AODV is a popular routing protocol for wireless ad hoc networks, first introduced in [41], and later revised for several times.

In this algorithm, routes are constructed dynamically whenever requested. Every node has its own routing table to maintain information about the routes of the received packets. When a node receives a packet (whether it is a route discovery or data packet), it updates its own routing table to keep the shortest and freshest path to the source or destination of the received packet. Three different tables are used to store information about the neighbors, routes and received messages:

- neighbor table: keeps the adjacency states of the node's neighbors. The neighbor state can be one of the following values:
 - *Confirmed*: indicates that a bidirectional link to that neighbor exists. This state is achieved either through receiving a *rrep* message in response to a previously sent *rreq* message, or a *RREP_Ack* message as a response to a previously sent *rrep* message (requested an *RREP_Ack*) to that neighbor.

² <https://tools.ietf.org/html/draft-ietf-manet-aodvv2-11>

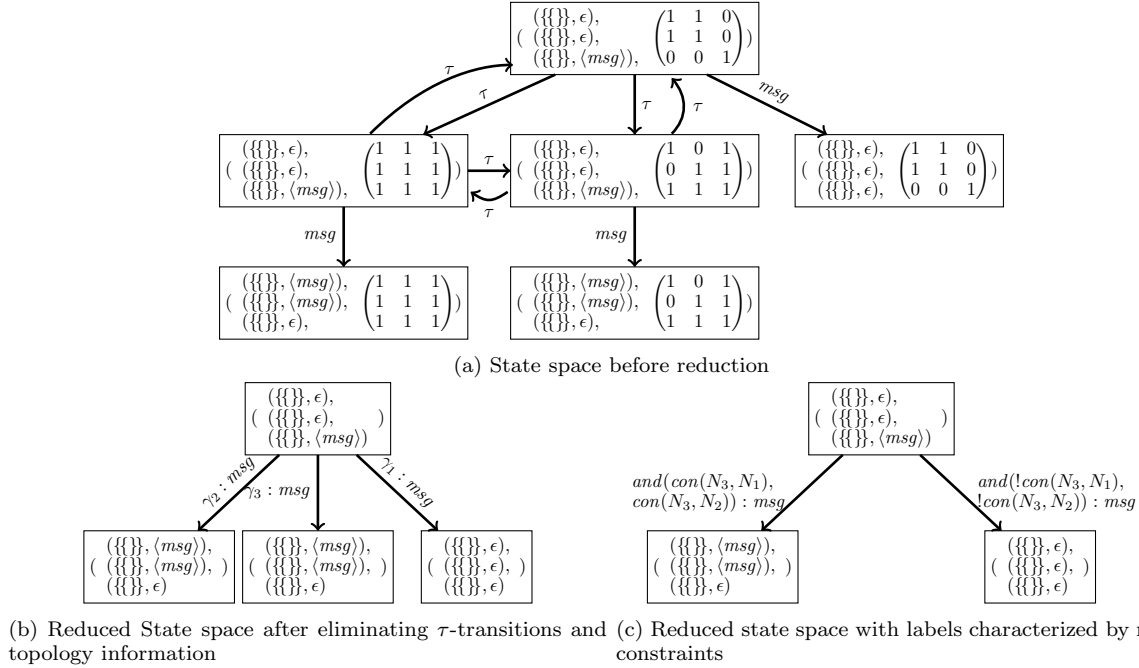


Fig. 11. State space before and after applying reduction

- *Unknown*: indicated that the link to that neighbor is currently unknown. Initially, all the state of the links to the neighbors are unknown.
- *Blacklisted*: indicates that the link to that neighbor is unidirectional. When a node has failed to receive the *RREP_Ack* message in response to its *rreq* message to that neighbour, the neighbor state is changed to blacklisted. Hence, it stops forwarding any message to it for an amount of time, *ResetTime*. After reaching the *ResetTime*, neighbor state would be set to unknown.
- route table: contains information about discovered routes and their status: The following information is maintained for each route:
 - *SeqNum*: destination sequence number
 - *route_state*: the state of the route to the destination which can have one of the following values:
 - *unconfirmed*: when the neighbor state of the next hop is unknown;
 - *active*: when the link to the next hop has been confirmed, and the route is currently used;
 - *idle*: when the link to the next hop has been confirmed, but it has not been used in the last *ACTIVE_INTERVAL*;
 - *invalid*: when the link to the next hop is broken, i.e., the neighbor state of the next hop is blacklisted.
 - *Metric*: indicates the cost or quality of the route, e.g., hop count, the number of hops to the destination
 - *NextHop*: IP of the next hop to the destination
 - *Precursors* (optional feature): the list of the nodes interested in the route to the destination, i.e., upstream neighbors.
- route message table, also known as *RteMsg Table*: contains information about previously received route messages such as *rreq* and *rrep*, so that we can determine whether the new received message is worth processing or not, it is redundant. Each entry of this table contains the following information:
 - *MessageType*: which can be either *rreq* or *rrep*

- *OrigAdd*: IP address of the originator
- *TargAdd*: IP address of the destination
- *OrigSeqNum*: sequence number of the originator
- *TargSeqNum*: sequence number of the destination
- *Metric*

When one node, i.e., source, intends to send a package to another, i.e., destination, it looks up its routing table for a valid route to that destination, a route which its route state is not invalid. If there is no such a route, it initiates a route discovery procedure by broadcasting a *rreq* message. The freshness of the requested route is indicated through the sequence number of the destination that the source is aware of. Whenever a node initiates a route discovery, it increases its own sequence number with the aim to define the freshness of its route request. Every node upon receiving this message checks its routing table for finding a route to the requested destination. If there is such a path or the receiver is in fact the destination, it informs the sender through unicasting a *rrep* message. However an acknowledgment is requested whenever the neighbor state of the next hop is *unconfirmed*. Otherwise, it re-broadcasts the *rreq* message to examine if any of its neighbors has a valid path. Meanwhile, a reverse forwarding path is constructed to the source over which *rrep* messages are going to be communicated later. In case a node receives a *rrep* message, if it is not the source, it forwards the *rrep* after updating its routing table with the received route information. Whenever a node fails to receive a requested *RREP_Ack*, it uses a *rerr* message to inform all its neighbors intended to use the broken link to forward their packets.

In our model, each node is represented through a rebe (actor), identified by an IP, with a routing table and a sequence number (*sn*). In addition, every node keeps track of the adjacency status to its neighbors, neighbor table, through *neigh_state* array, where *neigh_state*[*i*] = *true* indicates that it is adjacent to a node with the IP *i*, while *false* indicates to either its adjacency status is *unknown* or *blacklisted* (since timing issues are not taken into account, these two statuses are considered the same). As the destinations of any two arbitrary rows of a routing table are always different, so the routing table can have at most *n* rows, where *n* is the number of nodes in the model. Therefore, the routing table is modeled by the set of arrays, namely, *dsn*, *route_state*, *hops*, *nhops*, and *pres*, to represent *SeqNum*, *route_state*, *Metric*, *NextHop*, and *Precursors* columns of the routing table, respectively. The arrays *dsn* and *route_state* are of size *n*, while the arrays *hops*, *nhops*, and *pers* are of size *n* × *n*. For instance, *dsn*[*i*], keeps the sequence number of the destination with IP *i*, while *nhops*[*i*][*j*] contains the next hop of the *j*-th route to the destination with the IP *i*.

- *dsn*: destination sequence number
- *route_state*: an integer that refers to the state of the route to the destination and can have one of the following values:
 - *route_state*[*i*] = 0: the route is *unconfirmed*, there may be more than one route to the destination *i* with different next hops and hop counts;
 - *route_state*[*i*] = 1: the route is *valid*, the link to the next hop has been confirmed, the route state in the protocol is either **active** or **idle**, since we abstract from the timing issues, these two states are depicted as one;
 - *route_state*[*i*] = 2: the route is *invalid*, the link to the next hop is broken;
- *hops*: the number of hops to the destination for different routes
- *nhop*: IP of the next hop to the destination for different routes
- *pres*: an array that indicates which of the nodes are interested in the routes to the destination, for example *pres*[*i*][*j*] = *true* indicates that the node with the IP *j* is interested in the routes to the node with the IP *i*.

Since we have considered a row for each destination in our routing table, to indicate whether the node has any route to each destination until now, we initially set *dsn*[*i*] to *-1* which implies that the node has never had known any route to the node with the IP *i*. We refer to the all above mentioned arrays as *routing arrays*. Initially all integer cells of arrays are set to *-1* and all boolean cells are set to *false*. To model expunging a

route, its corresponding next hop and hop count entries in the arrays *nhops* and *hops* are set to -1 . Since we have only considered one node as the destination and one node as the source, the information in *rreq* and *rrep* messages has no conflict and consequently the route message table can be abstracted away. In other words, the routing table information can be used to identify whether the new received message has been seen before or not, as the stored routes towards the source represent information about *rreqs* and the routes towards the destination represent *rreps*.

Note that *rreq* and *rrep*, i.e., all route messages, carry route information to their source and destination, respectively. Therefore, a bidirectional path is constructed while these messages travel through the network. Whenever a node receives a route message, it processes incoming information to determine whether it offers any improvement to its known existing routes. Then, it updates its routing table accordingly in case of improvement. The processes of evaluating and updating the routing table are explained in the following.

6.1. Evaluating Route Messages

Every received route message contains a route and consequently is evaluated to check for any improvement. Note that a *rreq* message contains a route to its source while a *rrep* message contains a route to its destination. Therefore, as the routes are identified by their destinations (denoted by *des*), in the former case, the destination of the route is the originator of the message (i.e., $des = oip_-$) and in the latter, it is the destination of the message (i.e., $des = dip_-$). The routing table must be evaluated if one of the following conditions is realized:

1. no route to the destination has been existed, i.e., $dsn[des] = -1$
2. there are some routes to the destination, but all their route states are *unconfirmed*
3. there is a valid or invalid route to the destination in the routing table and one of following conditions holds:
 - the sequence number of the incoming route is greater than the existing one
 - the sequence number of the incoming route is equal to the existing one, however the hop count of the incoming route is less than the existing one (the new route offers a shorter path and also is loop free)

6.2. Updating the Routing Table

The routing table is updated as follows:

- if no route to the destination has been existed, i.e., $dsn[des] = -1$, the incoming route is added to the routing table.
- if the route states of the all existing routes to the destination are *unconfirmed*, the new route is added to the routing table.
- the incoming route has a different next hop from the existing one in the routing table, while the next hop's neighbor state of the incoming route is *unknown* and the route state of the existing route is *valid*. The new route should be added to the routing table since it may offer an improvement in the future and turn into *confirmed*.
- if the existing route state is *invalid* and the neighbor state of the next hop of the incoming route is *unknown*, the existing route should be updated with information of the received one.
- if the next hop's neighbor state of the incoming route is *confirmed*, the existing route is updated with new information and all other routes with the route state *unconfirmed* are expunged from the routing table.

As it was described earlier, there are three types of route discovery packets: *rreq*, *rrep* and *rerr*. There is a message server for handling each of these packet types:

- *rec_rreq* is responsible for processing a route discovery request message;
- *rec_rrep* handles a reply request message;
- *rec_rerr* updates the routing table in case an error occurs over a path and informs the interested nodes about the broken link.

There are also two message servers for receiving and sending a data packet. All these message servers will be discussed thoroughly in the following subsections.

6.3. *rreq* message server

This message server processes a received route discovery request and reacts based on its routing table, shown in Fig. 14. The *rreq* message has the following parameters: *hops_* and *maxHop* as the number of hops and the maximum number of hops, *dsn_* as the destination sequence number, and *oip_*, *osn_*, *dip*, and *sip_* respectively refer to the IP and sequence number of the originator, and the IP of the destination, and the IP of the sender. Whenever a node receives a route request, i.e., *rec_rreq(hops_, dip_, dsn_, oip_, osn_, sip_, maxHop)* message, it checks incoming information with the aim to improve the existing route or introduce a new route to the destination, and then updates its routing table accordingly (See also Sections 6.1 and 6.2). During processing an *rreq* message, a backward route, from the destination to the originator, is built by manipulating the routing arrays with the index *oip_*. Similarly, while processing an *rrep* message, it constructs a forwarded route to the destination by addressing the routing arrays with the index *dip_*. Therefore, the procedure of evaluating the new route and updating the routing table would be the same for both *rreq* and *rrep* messages except for different indices *oip_* and *dip_*, respectively.

Updating the routing table: Fig. 12 depicts this procedure which includes both evaluating the incoming route and updating the routing table (The code is the body of *if*-part in the line 7 of Fig. 14). If no route exists to the destination, receiving information is used to update the routing table and generate discovery packets, lines (1-12). The route state would be set based on the neighbor status of the sender: if its neighbor status is *confirmed*, the route state would be set to *valid*, otherwise to *unconfirmed*. The next hop is set to the sender of the message, i.e., *nhop[oip_][0] = sip_*. If a route exists to the destination (i.e., *oip_*), one of the following conditions happens:

- the route state is *unconfirmed*, lines (15-44): either it updates the routing table if there is a route with a next hop equal to the sender, or adds the incoming route to the first empty cells of *nhop* and *hops* arrays. If the neighbor status of the sender is *confirmed*, the all other routes with the same destination are expunged while the route state is set to *valid*, lines (28-37).
- the route state is *invalid* or it is *valid*, but the neighbor status of the sender is *confirmed*, lines (48-60): if the incoming message contains a greater sequence number, or an equal sequence number with a lower hop count, then it updates the current route while a new discovery message is generated.
- the route state is *valid* and the neighbor status of the sender is *unknown*, lines (64-72): the incoming route is added to the routing table and a new discovery message is generated if it provides a fresher or shorter path.

In these cases, if a new discovery message should be generated (when the node has no route as fresh as the route request), the auxiliary boolean variable *gen_msg* is set to *true*. In Fig. 14, after updating the routing table, if a new message should be generated, indicated by *if (gen_msg = true)*, it rebroadcasts the *rreq* message with the increased hop count if the node is not the destination, lines (51-54). Otherwise, it increases its sequence number and replies to the next hop(s) toward the originator of the route request, *oip_*, based on its routing table. Before unicasting *rrep* messages, next hops toward the destination, *dip_*, and the sender are set as interested nodes to the route toward the originator, *oip_*, lines (17-23). It unicasts each *rrep* message to its next hops one by one until it gets an ack from one, lines (24-44), ack reception is modeled implicitly through successful delivery of unicast i.e., *succ* part. If it receives an ack, it updates the route state to *valid* and the neighbor status of the next hop to *confirmed* and stops unicasting *rrep* messages. If it doesn't receive an *RREP_Ack* message from the next hop when the route state is *valid*, it initiates the error recovery procedure.

Error Recovery Procedure: The code for this procedure is illustrated in Fig. 13 (its code is the body of *if*-part in the line 44 of Fig. 14). As explained earlier, this procedure is initiated when a node doesn't receive an *RREP_Ack* message from the next hop of the route with *valid* state. Then, it updates its route state to *invalid* and adds the sequence number of the originator to the array of invalidated sequence numbers, denoted by *dip_sqn*. Furthermore, it adds all the interested nodes in the current route to the list of affected neighbors, denoted by *affected_neighbours*, lines (3-7). It invalidates other valid routes that use the same broken next hop as their next hops, adds their sequence numbers to the invalidated array and sets the nodes

```

1  if(dsn[oip_]==-1) {
2      dsn[oip_]=osn_;
3      if(neigh_state[sip_]==true)
4          { route_state[oip_]=1; }
5      else
6          { route_state[oip_]=0; }
7      hops[oip_][0]=hops_;
8      nhop[oip_][0]=sip_;
9      gen_msg = true;
10 } else {
11     if(route_state[oip_]==0)
12     {
13         dsn[oip_]=osn_;
14         route_num = 0;
15         for(int i=0;i<4;i++)
16         {
17             if(nhop[oip_][i]==-1 || nhop[oip_][i]==sip_)
18             {
19                 route_num = i;
20                 break;
21             }
22         }
23         if(neigh_state[sip_]==true)
24         {
25             route_state[oip_]=1;
26             for(int i=0;i<4;i++)
27             {
28                 hops[oip_][i]=-1;
29                 nhop[oip_][i]=-1;
30             }
31             hops[oip_][0]=hops_;
32             nhop[oip_][0]=sip_;
33         }
34         else {
35             route_state[oip_]=0;
36             hops[oip_][route_num]=hops_;
37             nhop[oip_][route_num]=sip_;
38         }
39     }
40     else {
41         if(route_state[oip_]==2 || neigh_state[sip_]==true)
42         {
43             /* update the existing route */
44             if((dsn[oip_]==osn_ && hops[oip_][0]>hops_) || dsn[oip_]<osn_)
45             {
46                 dsn[oip_]=osn_;
47                 if(neigh_state[sip_]==true)
48                     { route_state[oip_]=1; }
49                 else
50                     { route_state[oip_]=0; }
51                 hops[oip_][0]=hops_;
52                 nhop[oip_][0]=sip_;
53                 gen_msg = true;
54             }
55         }
56         else {
57             int index=-1;
58             for(int i=0;i<4;i++)
59             {
60                 if(nhop[oip_][i]==sip_)
61                     index=i;
62             }
63             if(nhop[oip_][index]==-1)
64             {
65                 if((dsn[oip_]==osn_ && hops[oip_][0]>hops_) || dsn[oip_]<osn_)
66                 {
67                     dsn[oip_]=osn_;
68                     hops[oip_][index]=hops_;
69                     nhop[oip_][index]=sip_;
70                     gen_msg = true;
71                 }
72             }
73         }
74     }
75 }

```

```

1  if (route_state[oip_] == 1)
2  {
3      route_state[oip_] = 2;
4      dip_sqn[oip_] = dsn[oip_];
5      for (int k=0; k<4; k++)
6      {
7          if (pre[oip_][k] == true)
8              { affected_neighbours[k] = true; }
9      }
10     for (int j=0; j<4; j++)
11     {
12         for (int r=0; r<4; r++)
13         {
14             if (nhop[oip_][r] != -1 && nhop[j][0] == nhop[oip_][r])
15             {
16                 route_state[j] = 2;
17                 dip_sqn[j] = dsn[j];
18                 for (int k=0; k<4; k++)
19                 {
20                     if (pre[j][k] == true)
21                         { affected_neighbours[k] = true; }
22                 }
23                 break;
24             }
25         }
26     }
27     multicast(affected_neighbours, rec_rerr(dip_, ip, dip_sqn));
28 }

```

Fig. 13. The error recovery procedure

interested in those routes as affected neighbors, lines (8-26). Finally, it multicasts an *rerr* message which contains the destination IP, the node IP, and the invalidated sequence numbers to the affected neighbors, line 27.

6.4. rrep message server

This message server, shown in Fig. 15, processes the received reply messages and also constructs the route forward to the destination. At first, it updates the routing table and decides whether the message is worth processing, as previously mentioned for *rreq* messages, and constructs the route, but this time to the destination (its code is similar to the one in Fig. 12 except that *dip_* is used instead of *oip_*, and is placed at the line 6 of Fig. 15). This message is sent backwards till it reaches the source through the reversed path constructed while broadcasting the *rreq* messages. When it reaches the source, it can start forwarding data to the destination. In case the node is not the originator of the route discovery message, it updates the array of interested nodes, lines (17-23). Then, it unicasts the message to the next hop(s), on the reverse path to the originator, lines (24-42). Based on the AODVv2 protocol if connectivity to the next hop on the route to the originator is not confirmed yet, node must request a Route Reply Acknowledgment (*RREP_Ack*) from the intended next hop router. If a *RREP_Ack* is received, then the neighbor status of the next hop and route state must be updated to *confirmed* and *valid*, lines (30-36), respectively, otherwise the neighbor status of the next hop remains *unknown*, lines (37-40). This procedure is modeled through *conditional unicast* which enables the model to react based on the delivery status of the unicast message so that *succ* models the part where the *RREP_Ack* is received while *unsucc* models the part where it fails to receive an acknowledgment from the next hop. In case the unicast is unsuccessful and the route state is valid, the error recovery procedure will be followed lines (43-46).

```

1  msgsrv rec_rreq(int hops_,int dip_ ,int dsn_ ,int oip_ ,int osn_ ,int sip_ ,int maxHop)
2  {
3      int [] dip_sqn=new int[4];
4      int route_num;
5      bool [] affected_neighbours=new bool[4];
6      bool gen_msg = false;
7      if(ip!=oip_)
8      {
9          //evaluate and update the routing table
10     }
11     if(gen_msg==true)
12     {
13         if(ip==dip_)
14         {
15             bool su = false;
16             pre[dip_][sip_]=true;
17             for(int i=0;i<4;i++)
18             {
19                 int nh = nhop[dip_][i];
20                 if(nh!=-1)
21                     { pre[oip_][nh]=true; }
22             }
23             for(int i=0;i<4;i++)
24             {
25                 if(nhop[oip_][i]!=-1)
26                 {
27                     int n_hop = nhop[oip_][i];
28                     sn = sn+1;
29                     /* unicast a RREP towards oip of the RREQ */
30                     unicast(n_hop,rec_rrep(0 , dip_ , sn , oip_ , self))
31                     succ:
32                     {
33                         route_state[oip_]=1;
34                         neigh_state[n_hop]=true;
35                         su = true;
36                         break;
37                     }
38                     unsucc:
39                     {
40                         neigh_state[n_hop]=false;
41                     }
42                 }
43             }
44             if(su==false && route_state[oip_]==1)
45             {
46                 /* error recovery procedure */
47             }
48         }
49         else {
50             hops_ = hops_+1;
51             if(hops_<maxHop)
52                 { rec_rreq(hops_,dip_,dsn_,oip_,osn_, self ,maxHop); }
53         }
54     }
55 }

```

Fig. 14. The rreq message server


```

1 msgsrv rec_rrep(int hops_,int dip_,int dsn_,int oip_,int sip_) {
2   int [] dip_sqn=new int[4];
3   bool [] affected_neighbours=new bool[4];
4   bool gen_msg = false;
5   int n_hop,route_num;
6   /*evaluate and update the routing table */
7   if(gen_msg==true)
8   {
9       if(ip==oip_)
10      {
11          /* this node is the originator of the corresponding RREQ */
12          /* a data packet may now be sent */
13      }
14      else {
15          hops_ = hops_+1;
16          bool su = false;
17          pre[dip_][sip_]=true;
18          for(int i=0;i<4;i++)
19          {
20              int nh = nhop[dip_][i];
21              if(nh!=-1)
22                  { pre[oip_][nh]=true; }
23          }
24          for(int i=0;i<4;i++)
25          {
26              if(nhop[oip_][i]!=-1)
27              {
28                  n_hop = nhop[oip_][i];
29                  unicast(n_hop,rec_rrep(hops_,dip_,dsn_,oip_, self))
30                  succ:
31                  {
32                      route_state[oip_]=1;
33                      neigh_state[n_hop]=true;
34                      su = true;
35                      break;
36                  }
37                  unsucc:
38                  {
39                      neigh_state[n_hop]=false;
40                  }
41              }
42          }
43          if(su==false && route_state[oip_]==1)
44          {
45              /* error recovery procedure */
46          }
47      }
48  }
49 }

```

Fig. 15. The rrep message server

6.5. rerr message server

This message server, shown in Fig. 16, processes the received error messages and informs those nodes that depend on the broken link. When a node receives an *rerr* message, it must invalidate those routes using the broken link as their next hops and sends the *rerr* message to those nodes interested in the invalidated routes. This message has only two parameters: *sip_* which indicates the IP of the sender, and *rip_rsn*, which contains the sequence number of those destinations which are become unaccessible from the *sip_*.

For all the *valid* routes to the different destinations, it examines whether the next hop of the route to the destination is equal to *sip_* and the sequence number of the route is less then the received sequence number, line 10. In case the above conditions are satisfied, the route is invalidated, lines (11-19), and *rerr* message is sent to the affected nodes, line 21.

```

1 msgsrv rec_rerr(int source_, int sip_, int [] rip_rsn) {
2     int [] dip_sqn=new int[4];
3     bool [] affected_neighbours=new bool[4];
4     if (ip!=source_)
5     {
6         //regenerate rrer for invalidated routes
7         for (int i=0;i<4;i++)
8         {
9             int rsn=rip_rsn[i];
10            if (route_state[i]==1 && nhop[i][0]==sip_ && dsn[i]<rsn && rsn!=0)
11            {
12                route_state[i]= 2;
13                dip_sqn[i]=dsn[i];
14                for (int j=0;j<4;j++)
15                {
16                    if (pre[i][j]==true)
17                    { affected_neighbours[j]=true; }
18                }
19            }
20        }
21        multicast(affected_neighbours, rec_rerr (source_, self ,dip_sqn));
22    }
23 }

```

Fig. 16. The rerr message server

6.6. newpkt message server

Whenever a node intends to send a data packet, it creates a *rec_newpkt* which has only two parameters, *data* and *dip_*. The code for this message server is shown in Fig. 17. If it is the destination of the message, it delivers the message to itself, lines (4-7). Otherwise, if it has a valid route to the destination, it sends data using that route, lines (12-16). If it has no valid route, it increases its own sequence number and broadcasts a route request message, lines (18-26). In addition, if a route to the destination is not found within *RREQ_WAIT_TIME*, node retries to send a new *rreq* message after increasing its own sequence number. Since we abstracted from time, we model this procedure through *resend_rreq* message server which attempts to resend *rreq* message while the node sequence number is less than 3 (to make the state space finite).

7. Evaluation

In this section, we will review the results obtained from constructing the efficient state spaces for the two introduced wRebeca models, the flooding and AODV protocols. Also, we briefly introduce our tool and its capabilities. Then, the loop freedom invariant is defined and one possible loop scenario is demonstrated. Finally, two properties that must hold for the AODV protocol are expressed that can be checked over the AODV model.

7.1. State Space Generation

Static Network . It is a network with a static topology, in other words the network constraint is defined so that it leads to only one valid topology. We illustrate the applicability of our counting abstraction technique on the flooding routing protocol. In contrast to the intermediate nodes (ones except the source and destination), the two source and destination nodes cannot be aggregated (due to their local states) with other nodes. However, in the case of the AODV protocol, no two nodes can be counted together due to the unique variables of IP and routing table of nodes. As the number of intermediate nodes with the same neighbors increases, the more reduction would take place. We have precisely chosen four fully connected network topologies to show the power of our reduction technique when the intermediate nodes increase from one to four.

```

1 msgsrv rec_newpkt(int data,int dip_) {
2   int [] dip_sqn=new int[4];
3   bool [] affected_neighbours=new bool[4];
4   if(ip==dip_)
5   {
6     /* the DATA packet is intended for this node */
7   }
8   else{
9     /* the DATA packet is not intended for this node */
10    store[dip_]=data;
11    if(route_state[dip_]==1)
12    {
13      /* valid route to dip_*/
14      /* forward packet */
15    }
16    else{
17      /* no valid route to dip_*/
18      /*send a new rout discovery request*/
19      if(sn<3)
20      {
21        sn++;
22        unicast( self ,resend_rreq(dip_));
23        rec_rreq(0,dip_,dsn[dip_], self ,sn, self ,4);
24      }
25    }
26  }
27 }
28 msgsrv resend_rreq(int dip_)
29 {
30   if(sn<3)
31   {
32     sn++;
33     unicast( self ,resend_rreq(dip_));
34     rec_rreq(0,dip_,dsn[dip_], self ,sn, self ,4);
35   }
36 }

```

Fig. 17. The rec_newpkt message server

Table 2. Comparing the size of state spaces with/without applying counter abstraction reduction

No. of intermidate nodes	No. of states before reduction	No. of states after reduction	No. of transitions before reduction	No. of transitions after reduction
1	11	11	15	15
2	132	95	339	215
3	2,390	845	8,869	2,499
4	56,426	7,355	270,209	26,134

Table 2 illustrates the number of states for running flooding protocol in different networks with different topologies before and after applying counter abstraction reduction. In the first, second, third, and fourth topology, there are three nodes with one intermediate, four nodes with two, five nodes with three, and six with four intermediates, respectively. By applying counter abstraction reduction, the intermediate nodes are collapsed together as they have the same role in the protocol. However, the effectiveness of this technique depends on the network topology and the modeled protocol.

Dynamic network. At these networks, topology is constantly changing, in other words there are more than one possible topology. The resulting state spaces after and before eliminating τ -transitions are compared for the two case studies while the topology is constantly changing for a networks of 4 and 5 nodes, as shown in Table 3. Table 4 depicts the constraints used to generate the state spaces and the number of topologies that each constraint results in. Constraints are chosen randomly here, just to show the effectiveness of our reduction technique. To this aim, we have randomly removed a (fixed) link from the network constraints. Nevertheless, constraints can be chosen wisely to limit the network topologies to those which are prone to

Table 3. Comparing the size of state spaces with/without applying τ -transition elimination reduction

	No. of nodes	No. of valid topologies	No. of states before eliminating τ	No. of transitions before eliminating τ	No. of states after eliminating τ	No. of transitions after eliminating τ
flooding protocol	4	4	2,312	12,924	578	1,781
	4	8	4,920	47,192	615	1,905
	4	16	10,240	179,904	640	2,160
	4	32	20,480	687,488	640	2,480
	4	64	44,480	2,917,696	695	3,113
AODV protocol	4	4	2,840	15,508	710	1,862
	4	8	11,632	106,440	1,454	3,627
	4	16	34,752	586,016	2,172	5,478
	4	32	102,848	3,420,544	3,215	10,852
	4	64	123,240	7,951,616	3,866	64,504
	5	16	-	-	110,095	402,030

Table 4. Applied network constraints

No. of nodes	No. of valid topologies	constraint
4	4	$and(and(con(node0, node1), con(node0, node3)), and(con(node2, node3), con(node1, node3)))$
4	8	$and(and(con(node0, node1), con(node0, node3)), con(node2, node3))$
4	16	$and(con(node0, node1), con(node2, node3))$
4	32	$con(node0, node1)$
5	16	$and(and(con(node0, node1), and(con(node0, node3), con(node4, node1))), and(con(node2, node3), and(con(node1, node3), con(node2, node4))))$

lead to an erroneous situation, i.e., violation of a correctness property like loop freedom. However, it is also possible to check the model against all possible topologies by not defining any constraint. In other words, at first modeler can focus on some suspicious network topologies and after resolving the raised issues it can check the model for all possible topologies. There are also some networks which have certain constraints about how topology can change, e.g., node 1 can never get into the communication range of node 2. These restrictions on topology changes can be reflected through constraints too. The size of state spaces are compared under different network constraints resulting different numbers of valid topologies. Eliminating τ -transitions and topology information manifestly decline the number of states and transitions even when all possible topologies are not restricted. Therefore, it makes MANET protocol verification possible in an efficient manner. Note that in case the size of the network was increased from four to five, we couldn't generate its state space in a short time (i.e., one day) before applying reduction (It took less than an hour to generate the reduced state space).

7.2. Tool Support

The presented modeling language is supported by a tool (available at [6]), providing a number of options to generate the state space. A screen-shot of this tool is given in Fig. 18. This tool supports both bRebeca and wRebeca models characterized by different file types. After opening a model, the tool extracts the information of the reactive classes, such as the state variables and message servers, and also the main part including the rebec declarations and the network constraint. Then it generates several classes in Java language based on the obtained information and compiles them together with some abstract and base classes (common in all models), for example *global state* and *topology*, to build an engine that constructs the model state space upon its execution. Before compiling, user can decide about rebecs message processing method, in a FIFO manner (queue) or in an arbitrary way (bag), and if the reduction should be applied. To take advantage of all hardware capabilities, we have implemented our state space generation algorithm in a multi-threaded way to leverage the power of multi-core CPUs.

During state space generation, information about the state variables and transitions are stored as an LTS in the Aldebaran format. This LTS can be evaluated by tools such as *mCRL2* toolset [2]. For example, one can express desired properties in μ -calculus [33] and verify them. Also, as explained in Section 5, labels are

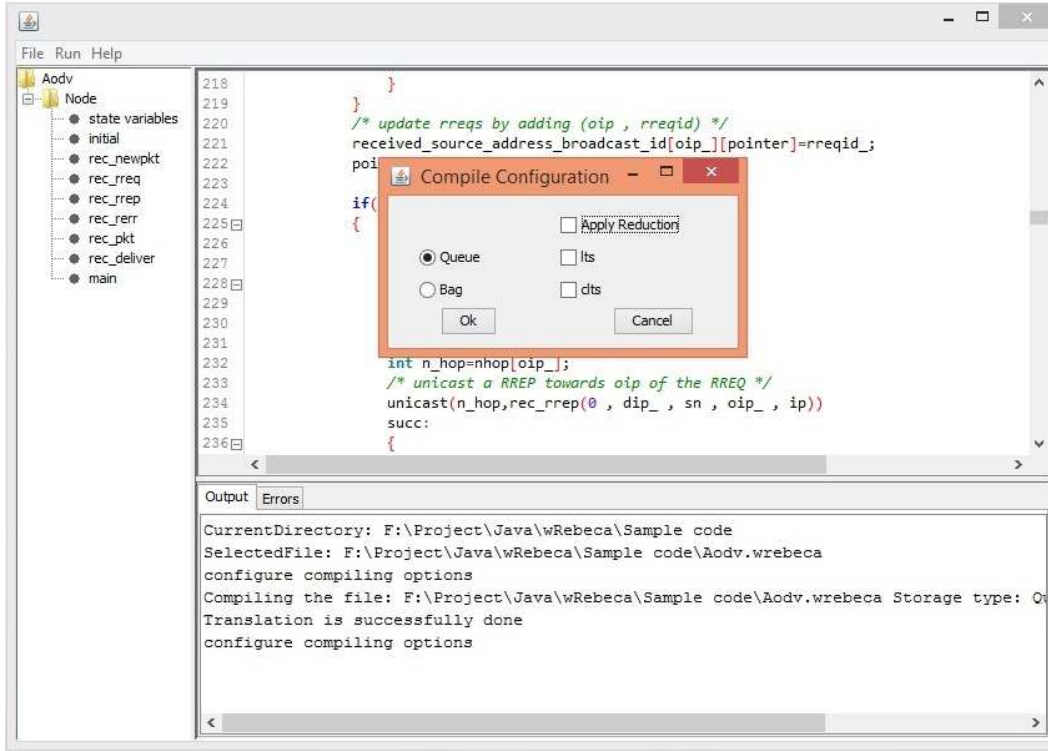


Fig. 18. An screen-shot of wRebeca tool with *compilation info* window to configure the state space generator

extended with network constraints as defined in [20] so that the reduced LTS can be model checked with respect to underlying topology [21].

7.3. Model Checking of the AODV Protocol Properties

There are different ways to check a given property on a wRebeca model. Invariant properties can be evaluated while generating the state space by checking each reached global state against defined invariants. Furthermore, the resulting state space can be model checked by tools supporting Aldebaran format such as mCRL2 and CLTS model checker.

7.3.1. Checking the loop freedom invariant

Loop freedom is one of the well-known property which must hold for all routing protocols, i.e., the AODV protocol. For example, consider the routes to the destination x in the routing tables of all nodes, where $node_0$ has a route to x with the next hop $node_1$, $node_1$ has a route to x with the next hop $node_2$, and $node_2$ has a route to x with the next hop $node_0$. The given example constructs a loop which consists of the three nodes, $node_0$, $node_1$, and $node_2$. A state is considered *loop free* if the collective routing table entries of all nodes for each pair of a source and destination do not form a loop. As it was mentioned earlier in AODV-v2-11, each route may have more than one next hop when the adjacency states of the next hops are *unconfirmed*. Therefore, while the loop freedom of a state is checked, one must take into account all next hops stored for each route. Then, for each next hop checks whether it leads to a loop or not. A routing protocol deployed on a network is called loop free, if all of its states be loop free. In other words, loop freedom property of a protocol is an invariant (which can be easily specified by the ACTL-X fragment of μ -calculus, and hence, is preserved by the reduced semantic model). However, we extend our state space generator engine (produced by our tool) to check the loop freedom property of each newly generated global state on the fly. To this aim, we specified a recursive function to determine whether in a global state the next-hops in different nodes collectively would

```

1 | bool loop_freedom(des:int, cur:int, visited:Set<int>){
2 |     for(int i=0; i<n; i++){
3 |         if ((state.node(cur).nhops[des][i]!=-1) && (!visited.contains(state.node(cur).nhops[des][i]))
4 |             && loop_freedom(des,i,visited.add(i)))
5 |             return true;
6 |         else
7 |             return false;
8 |     }

```

Fig. 19. Checking loop freedom property on a global state: we have used a dot notation to access the array *nhops* of the rebec with the identifier *i*, i.e., *state.node(i)*, where *state* is the newly generated global state

lead to a loop scenario, as shown in Fig. 19, as a part of the state generator class. Whenever a new state is reached before proceeding any further, it is checked by calling *loop_freedom*(4,1,new Set < int > (1)) and *loop_freedom*(1,4,new Set < int > (4)) as *node₄* and *node₁* are the destination and source of our model to assure that no loop would be formed on the forward/backward routes between the source and destination. If loop freedom condition is violated, *loop_freedom* function returns *false*, the state generator engine doesn't process the new global state and returns it and one path which has led to that global state. The function *loop_freedom* has three parameters: *des* refers to the destination of the route, *cur* refers to IP of the current node which is going to be processed and *visited* is the list of IPs of those nodes which have been processed.

Although keeping more than one next hop for each route may increase the route availability, it compromises its validity by violating the loop freedom invariant in a network of at least four nodes with a dynamic topology. Consider the network topology shown in Fig. 2a. The following scenario explains steps that lead to the invariant violation ³:

1. *node₂* initiates a route discovery procedure for destination *node₃* by broadcasting a *rreq* message.
2. *node₁* and *node₄* upon receiving the *rreq* message, add a route to their routing tables towards *node₂* and store *node₂* as their next hop. Since it is the first time that these nodes have received a message from *node₂*, the neighbor state of *node₂* is set to *unconfirmed*. Therefore, the route state is *unconfirmed*.
3. As *node₁* and *node₄* are not the intended destination of the route request, they rebroadcast the *rreq* message.
4. *node₁* receives the *rreq* message sent by *node₄* and since the route to *node₂* is *unconfirmed* it adds *node₄* as a new next hop to *node₂*.
5. *node₄* also adds *node₁* as the new next hop towards *node₂* after processing the *rreq* sent by *node₁*. At this point a loop is formed between *node₁* and *node₄*.
6. *node₃* receives the *rreq* message sent by *node₁* and since it is the destination, it sends a *rrep* message towards *node₁*.
7. *node₂* moves out of *node₁* and *node₄* communication ranges.
8. *node₁* receives the *rrep* message sent by *node₃* and as the route state towards *node₂* is *unconfirmed* it multicasts the *rrep* message to the existing next hops, *node₂* and *node₄*. Since *node₄* is adjacent to *node₁*, it receives the message and then sends an ack to *node₁*. Therefore, *node₁* sets the neighbor state of *node₄* to *confirmed* and subsequently the route state towards *node₂* to *valid*. Then expunge the next hop *node₂* which has not received an ack from.
9. *node₄* by receiving the *rrep* message from *node₁* multicasts it to its next hops towards *node₁* and *node₂*, and similar to *node₁*. It updates its routing table by validating *node₁* as its next hop to *node₂*.

We have found the scenario in the wRebeca model with the network constraint resulting four topologies as indicated in Table 4. However, this scenario was also found for all the network constraints described in the Table. Furthermore, we can generalize the scenario to all networks with the same connectivity when the communications occur, and the same mobility scenario.

³ We have communicated this scenario with the AODV group and they have confirmed it. In response, their route information evaluation was modified, published in version 13 of the draft (<https://tools.ietf.org/html/draft-ietf-manet-aodvv2-13>).

$$\begin{aligned}
& \forall x, y : \mathbb{N}((x > 0 \wedge x < 4 \wedge y > 0 \wedge y < x) \Rightarrow [src_sn(x).true^*.src_sn(y)]false) \\
& \forall x, y, m, n : \mathbb{N}(x \geq 0 \wedge x < 4 \wedge y \geq 0 \wedge y < 4 \wedge \\
& \quad m \geq 0 \wedge m < 4 \wedge n \geq 0 \wedge n < 4 \wedge \\
& \quad (m < x \vee n < y)) \Rightarrow [true^*.info_i_dsn(x, y).true^*.info_i_dsn(m, n)]false)
\end{aligned}$$

Fig. 20. μ -calculus properties verified by mCRL2

7.3.2. Checking the properties by mCRL2

Sequence numbers are used frequently by the AODV protocol to evaluate the freshness of routes. Therefore, it is important that each node's sequence number increases monotonically. To this end, we manually configured the state generator to add two self-loops to each state with the label $src_sn(x)$ to monitor the sequence number of the source node, where x is sn of the source node, and the label $info_i_dsn(y, z)$ to trace the destination sequence number of routes to the source and destination for each node i (i.e., the backward and forward routes to the destination of our model), where y and z are $dsn[src]$ and $dsn[dst]$ of node i , respectively. These properties are expressed through the ACTL-X fragment of μ -calculus as shown in Fig. 20. The first formula asserts monotonic increase of the source sequence number. The second formula assures the destination sequence numbers stored in the routing table of $node_i$ are increased monotonically, and must hold for the all nodes in the model.

7.3.3. Checking packet delivery property by the CLTS model checker

The CLTS model checker can be used to express and verify interesting properties of MANET protocols dependent to the underlying topology specified in Constrained Action Computation Tree Logic (CACTL) [21], an extension of Action CTL [12]. The path quantifier *All* in CACTL is parametrized by a multi-hop constrain over the topology, which specifies the pre-condition required for paths of a state to be inspected. Therefore, a state satisfies $\mathbf{A}^\mu \varphi$ if its paths over which the multi-hop constraint μ holds, also satisfy φ . It also contains the two temporal operators *until* and *weak until* to specify the path formulae $\phi_\chi \mathbf{U}_{\chi'} \phi'$ and $\phi_\chi \mathbf{W}_{\chi'} \phi'$ to denote a path over which states satisfying ϕ are met by actions of χ until a state satisfying ϕ' is met by actions of χ' (in case of weak until, the state satisfying ϕ' can never be met).

The important property of *packet delivery* in routing or information dissemination protocols in the context of MANETs becomes: if there exists an end-to-end route (multi-hop communication path) between two nodes A and C for a sufficiently long period of time, then packets sent by A will eventually be received by C [18]. To specify such the property, inspired from [18] we revised our specification to include data packet handling (to forward the packet to its next hop towards the destination) in addition to the route discovery packets and their corresponding handlers. Therefore, whenever a node, source, discovers a route to an intended destination, it starts forwarding its data packet through the next hop specified in its routing table. The data packet is forwarded by intermediate nodes to their next hops. When the data packet reaches the intended destination, it delivers the data to itself by unicasting the *deliver* message to itself. In case an intermediate node fails to forward the message, the error recovery procedure is followed as explained in Section 6. Consequently, using the following formula, we can verify packet delivery property:

$$\mathbf{A}^{true} (true \neg_{rec_newpkt(0,4)} \mathbf{W}_{rec_newpkt(0,4)} \mathbf{A}^{n_1 \dashrightarrow n_4 \wedge n_4 \dashrightarrow n_1} (true \tau \mathbf{U}_{deliver()} true))$$

It expresses that as long as there is a stable multi-hop path from n_1 to n_4 and viceversa (specified by $n_1 \dashrightarrow n_4 \wedge n_4 \dashrightarrow n_1$), any $rec_newpkt(0, 4)$ message is proceeded by a *deliver*() message after passing τ -transitions which abstract away from other message communications. By model checking the resulting CLTS of the AODVv2 model, we found a scenario in which the property does not hold. We explain this scenario in a network of three nodes N_1 , N_2 and N_3 , where node N_3 is always connected to the nodes N_1 and N_2 , while the connection between the nodes N_1 and N_2 is transient. Therefore, the mobility of nodes leads to the topologies shown in Fig. 10b and Fig. 10c. Assume the topology is initially as the one in Fig. 10b:

- Node N_1 unicasts a $rec_newpkt(data, N_2)$ to itself, indicating that it wants to send *data* to node N_2 .
- Node N_1 initiates a route discovery procedure by broadcasting an $rreq_{N_1,0}$ message to its neighbors, i.e., nodes N_3 and N_2 . Note that $rreq_{a,i}$ refers to an *rreq* message received from node a with the hop count of i . Each *rreq* message has more parameters but here only these two parameters are of interest and

the other parameters are assumed to be equal for all the *rreq* messages, i.e., the destination and source sequence numbers, and the source and destination IPs.

- Node N_3 processes the $rreq_{N_1,0}$ and since it is not the destination and has no route to N_2 in its routing table, rebroadcasts the $rreq_{N_3,1}$ message to its neighbors, nodes N_1 and N_2 , after increasing the hop count. At this point, node N_2 has two messages in its queue, $rreq_{N_1,0}$ and $rreq_{N_3,1}$.
- Node N_1 moves out of the communication range of node N_2 , resulting the network topology shown in Fig. 10c.
- Node N_2 takes $rreq_{N_1,0}$ from the head of its queue and updates its routing table by setting N_1 as the next hop in the route towards N_1 . As node N_2 is the intended destination for the route discovery message, it unicasts an *rrep* message towards the originator, N_1 , indicating that the route has been built and it can start forwarding the data. Therefore, node N_2 attempts to unicast an *rrep* message to node N_1 , i.e., its next hop towards the originator.
- Since the connection between the nodes N_1 and N_2 is broken, it fails to receive an ack from N_1 and marks the route as *invalid*.
- Node N_2 takes $rreq_{N_3,1}$ from its queue and since the route state towards N_1 is *invalid*, it evaluates the received route to determine whether it is loop free. Updating the routing table with the received route is said to be “loop free”, if the received message cost, e.g., the hop count is less than or equal to the existing route cost. Since the hop count of the received message is greater than the existing one, it does not update the existing route and the message is discarded.

Although the route through node N_3 to node N_1 seems to be valid, the protocol refuses to employ it to prevent possible loop formation in the future.

8. Related Work

A large number of studies has been conducted for modeling and verification of MANETs protocol using different approaches to tackle its specific challenges. These challenges, as discussed in Section 3, are modeling the underlying topology, mobility and local broadcast.

Several process algebra modeling frameworks have been proposed for broadcasting environments and particularly MANETs such as CBS# [39], CWS [36], CMN [35], the ω -calculus [50], $bA\pi$ [25], CMAN [24, 23], RBPT [19] and the broadcast psi-calculi [9, 44]. Each of these proposed frameworks overcome the modeling difficulties such as local broadcast, data handling, and message delivery guarantee in different ways. They usually truly overcome one or two of these challenges while the others remain unresolved. All these approaches (except [23, 44]) suffer from lack of message delivery guarantee that makes them inappropriate for analyzing properties like packet delivery [18]. They model broadcast through either an enforced synchronized or lossy communication. All these languages, except CBS# and CWS, assume that communications are lossy so that although a node is in the transmission range of a sender, but it may not receive the message. CBS# and CWS use enforced synchronization for broadcast to make sure that all ready nodes within the transmission range of a sender will receive the message. Although they guarantee message delivery to the ready receivers, it is not possible to define meaningful nodes in their syntax which are always ready [18]. Almost all these languages model mobility of nodes in their semantics through arbitrary changes of the topology with the exception that it is modeled through different generations of assertions on connectivity information in [9]. In [18] the process algebra AWN is proposed particularly for modeling wireless mesh network (WMN) routing protocols which uses local broadcast with message delivery guarantee. It defines its own data structures to model routing tables and other necessary data types to model the AODV protocol. In addition, conditional unicast is introduced for modeling the procedure to act based on the message delivery acknowledgment. In all these approaches, while a node is busy processing a message, it fails to receive messages from other nodes. Therefore, either nodes are defined to be input enabled as in CBS# and CWS or a process with a queue that concurrently stores new messages should be specified as in AWN. In contrast, in wRebeca, communications are asynchronous and received messages are stored in queues implicitly at the semantic level. This enables us to easily exploit the counting abstraction technique.

There are different approaches [48, 15, 1, 37] with the aim to analyze networks with an infinite number of nodes, where nodes execute an instance of a network process. A network configuration is represented as a graph in which each individual node represents a state of the process. The behavior of a process is

modeled by an automaton. The network configuration transforms either due to the process evolution at a network node or the topology reconfiguration. Verification of safety properties, reaching to an undesirable configuration starting from an initial configuration, is parameterized due to any possible number of nodes and connectivity among them. It is proved that the problem of parameterized safety properties, so called *control states reachability problem*, is undecidable. However, the problem turns out to be decidable for the class of bounded path graphs [15, 14]. Decidability of the problem is discussed when configurations evolve due to discrete/continuous clocks at processes [1]. As we restrict our models to the ones with a fixed and known number of nodes, they boil down to the finite state problems. Furthermore, an inductive approach based on reduction to prove compositional invariants for the dynamic networks is presented in [37]. Their symmetry reduction exploits localized neighborhood symmetries defined in terms of local states of nodes and their behaviors. This approach was used to prove loop freedom of AODVv2-04 [38] for an arbitrary number of nodes. Another approach is based on symbolic backward reachability analysis to reason about MANETs which is not guaranteed to terminate due to the undecidability of problem [48]. While these approaches are scalable to prove a property for MANETs, our approach is valuable to examine confirmation and diagnostics of suspected errors in the early phase of protocol development for a limited number of nodes. In other words, our efficient model checking tool can be used as an initial step before involving to generalize a property for an arbitrary sized network.

Another approach for modeling MANET protocol is through using the existing formalisms like SPIN [13, 54], UPPAAL [17, 34, 55]. In a SPIN model, nodes connectivity are modeled with the help of PROMELA channels, one per each node. Also, mobility is modeled by *case selection* instruction provided by PROMELA, for modeling nondeterminism. In the initialization section, possible links to other neighbors are defined as different *cases* that all will be checked for a model. Since it does not provide a specific technique to reduce the state space, its state space grows very fast and it is not feasible to check all possible topologies. Therefore, models would be limited to use the less number of topologies. In UPPAAL, connectivity is modeled through the set of arrays of booleans, while changing topology is modeled by a separate automaton which manipulates the arrays. In [54], a case study carried out to evaluate two model checkers, SPIN and UPPAAL. However, it was impossible to check all possible topologies and their arbitrary changes due to state space explosion, thus they had to narrow it down to some special mobility scenarios (as a part of the specification). However, our reduction technique makes it possible to verify a MANET for all possible topology changes.

9. Conclusion and Future Work

In this paper we extended the syntax and semantics of bRebeca, the actor-based modeling language for broadcasting environment, to support wireless communication in a dynamic environment. We addressed the key features of wireless ad hoc networks, namely reliable local broadcast, conditional unicast, and last but not least mobility. The reliable asynchronous local broadcast/unicast communication, and implicit support of message storages make our framework suitable to analyze MANETs with respect to different mobility scenarios. A modeler only focuses on how to decompose a protocol into a set of communicating actors to cover functionalities of the protocol under investigation.

To overcome the state space explosion, we leveraged counter abstraction technique to analyze ad hoc networks with static topologies. Our reduction technique performs well on protocols with no specific state variable that distinguishes each rebec, and topologies with many topologically equivalent nodes. We demonstrated effectiveness of our approach on the flooding protocol in different network settings. However, mobility ruins the soundness of our counting abstraction. To this end, we eliminated τ -transitions while topology information was removed from global states to considerably reduce the size of the state space. We integrated the proposed reduction techniques into a tool customizable to verify wRebeca models for different message storage policies and topology dynamism. Invariants can be checked during the state space generation while the resulting output can be fed into the existing model checking tools such as mCRL2 and CLTS model checker.

We presented a complete and accurate model of the core functionalities of a recent version of AODVv2 protocol (version 11). We abstracted optional features and timing aspects to make our model manageable. We verified loop freedom property in AODVv2-11 and found a scenario in which the property violated. Loop freedom has already been proved in [8, 18, 37] on various versions of AODV. The new version differs in the following aspects which distinguish our attempt: in this version multiple next hops are maintained for each destination and consequently the process to update the routing table is completely different; Different

statuses are considered for a route in the table of a node regarding the neighbor status of its next hop; Sequence numbers for invalid destinations in intermediate nodes are not increased anymore (like [37], in contrast to others). Although, these approaches focus on providing a general proof for the property, our model checking-based approach detects the error and the scenario that leads to it. Our approach, can be adopted to resolve conceptual/design errors in an iterative way in the early phase of protocol development. The positive result of verifications for a limited size of the network constitutes the induction base for all these approaches.

We plan to integrate our state-space generator tool into the verification environment [3] to take benefit of its model checker. Furthermore, we aim to run more case studies to extend application of our framework. To analyze real-time and probabilistic behavior of wireless network protocols, wRebeca can be extended in the same way of [29, 53]. To this aim, there is a need to examine the soundness of our reduction techniques when probability and time are introduced.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments on the earlier version of the paper, the AODV group for their supports, and Mohammad Reza Mousavi for his discussion on the paper.

References

- [1] Abdulla, P.A., Delzanno, G., Rezine, O., Sangnier, A., Traverso, R.: On the verification of timed ad hoc networks. In: 9th International Conference on Formal Modeling and Analysis of Timed Systems. LNCS, vol. 6919, pp. 256–270. Springer (2011)
- [2] mCRL2: analysing system behaviors. <http://www.mcrl2.org/>
- [3] Rebeca formal modeling language. <http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/Afra>
- [4] Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
- [5] Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Computer Aided Verification. pp. 64–78. Springer (2009)
- [6] wRebeca, Efficient Modeling of Mobile Ad hoc Networks. <http://fghassemi.adhoc.ir/wrebeca>
- [7] Bertsekas, D.P., Gallager, R.G.: Data Networks. Prentice Hall (1992)
- [8] Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4), 538–576 (2002)
- [9] Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Pohjola, J.-Å., Parrow, J.: Broadcast psi-calculi with an application to wireless protocols. Software and System Modeling 14(1), 201–216 (2015)
- [10] Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Computer Aided Verification. pp. 147–158. Springer (1998)
- [11] Cui, T., Chen, L., Ho, T.: Distributed optimization in wireless networks using broadcast advantage. In: Decision and Control. pp. 5839–5844. IEEE (2007)
- [12] De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Semantics of Systems of Concurrent Processes. Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer (1990)
- [13] De Renesse, R., Aghvami, A.: Formal verification of ad-hoc routing protocols using spin model checker. In: 12th IEEE Mediterranean, Electrotechnical Conference. vol. 3, pp. 1177–1182. IEEE (2004)
- [14] Delzanno, G., Sangnier, A., Traverso, R., Zavattaro, G.: On the complexity of parameterized reachability in re-configurable broadcast networks. In: Annual Conference on Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 18, pp. 289–300. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
- [15] Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of safety properties in ad hoc network protocols. In: First International Workshop on Process Algebra and Coordination. EPTCS, vol. 60, pp. 56–65 (2011)
- [16] Emerson, E.A., Treffer, R.J.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In: Correct Hardware Design and Verification Methods. pp. 142–156. Springer (1999)
- [17] Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.: Automated analysis of AODV using Uppaal. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 7214, pp. 173–187. Springer Berlin Heidelberg (2012)
- [18] Fehnker, A., Van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. arXiv preprint arXiv:1312.7645 (2013)
- [19] Ghassemi, F., Fokink, W., Movaghar, A.: Restricted broadcast process theory. In: Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM). pp. 345–354. IEEE Computer Society (2008)
- [20] Ghassemi, F., Fokink, W., Movaghar, A.: Verification of mobile ad hoc networks: An algebraic approach. Theoretical Computer Science 412(28), 3262–3282 (2011)

- [21] Ghassemi, F., Ahmadi, S., Fokkink, W., Movaghar, A.: Model checking MANETs with arbitrary mobility. In: *Fundamentals of Software Engineering*, pp. 217–232. Springer (2013)
- [22] van Glabbeek, R., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. *Journal of the ACM* 43(3), 555–600 (1996)
- [23] Godskesen, J.C.: A calculus for mobile ad-hoc networks with static location binding. *Electr. Notes Theor. Comput. Sci.* 242(1), 161–183 (2009)
- [24] Godskesen, J.: A calculus for mobile ad hoc networks. In: Murphy, A., Vitek, J. (eds.) *Coordination Models and Languages, LNCS*, vol. 4467, pp. 132–150. Springer Berlin Heidelberg (2007)
- [25] Godskesen, J.: Observables for mobile and wireless broadcasting systems. In: *Coordination Models and Languages, LNCS*, vol. 6116, pp. 1–15. Springer Berlin Heidelberg (2010)
- [26] Hewitt, C.: Viewing control structures as patterns of passing messages. *Artif. Intell.* 8(3), 323–364 (1977)
- [27] Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informatica* 47(1), 33–66 (2010)
- [28] Katoen, J.: Model checking: One can do much more than you think! In: *Fundamentals of Software Engineering*, pp. 1–14. Springer (2011)
- [29] Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Science of Computer Programming* 98, 184–204 (2015)
- [30] Kuhn, F., Lynch, N.A., Newport, C.C.: The abstract MAC layer. *Distributed Computing* 24(3-4), 187–206 (2011)
- [31] Javitani, S.P., Foster, C.C.: Finding an extremum in a network. In: *9th International Symposium on Computer Architecture*, pp. 321–325. ACM (1982)
- [32] Mahmud, S.A., Khan, S., Khan, S., Al-Raweshidy, H.: A comparison of manets and wmnns: commercial feasibility of community wireless networks and manets. In: *1st International Conference on Access Networks*. ACM (2006)
- [33] Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.* 46(3), 255–281 (2003)
- [34] McIver, A., Fehnker, A.: Formal techniques for the analysis of wireless networks. In: *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 263–270. IEEE (2006)
- [35] Merro, M.: An observational theory for mobile ad hoc networks (full version). *Information and Computation* 207(2), 194 – 208 (2009), special issue on Structural Operational Semantics (SOS)
- [36] Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. *Electronic Notes in Theoretical Computer Science* 158(0), 331 – 353 (2006)
- [37] Namjoshi, K.S., Treffer, R.J.: Analysis of dynamic process networks. In: *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, vol. 9035, pp. 164–178. Springer (2015)
- [38] Namjoshi, K.S., Treffer, R.J.: Loop freedom in aodvv2. In: *Formal Techniques for Distributed Objects, Components, and Systems*. LNCS, vol. 9039, pp. 98–112 (2015)
- [39] Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. *Theor. Comput. Sci.* 367(1-2), 203–227 (2006)
- [40] Peng, J.: A new arq scheme for reliable broadcasting in wireless lans. *IEEE Communications Letters* 12(2), 146–148 (2008)
- [41] Perkins, C.E., Belding-Royer, E.M.: Ad-hoc on-demand distance vector routing. In: *2nd Workshop on Mobile Computing Systems and Applications*, pp. 90–100. IEEE Computer Society (1999)
- [42] Plotkin, G.D.: A structural approach to operational semantics. *Tech. Rep. DAIMI FN-19*, University of Aarhus (1981)
- [43] Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infity)-Counter Abstraction. In: *14th International Conference on Computer Aided Verification*, pp. 107–122. CAV '02, Springer-Verlag (2002)
- [44] Pohjola, J.Å., Borgström, J., Parrow, J., Raabjerg, P.: Negative premises in applied process calculi. *Tech. rep.*, Department of Information Technology, Uppsala University (2013)
- [45] Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfssdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Sci. Comput. Program.* 89, 41–68 (2014)
- [46] Sabouri, H., Khosravi, R.: Delta modeling and model checking of product families. In: *Fundamentals of Software Engineering*, pp. 51–65. Springer (2013)
- [47] Sabouri, H., Sirjani, M.: Slicing-based reductions for rebeca. *Electronic Notes in Theoretical Computer Science* 260, 209–224 (2010)
- [48] Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, vol. 4963, pp. 18–32. Springer (2008)
- [49] Si, W., Li, C.: RMAC: A reliable multicast MAC protocol for wireless ad hoc networks. In: *33rd International Conference on Parallel Processing (ICPP 2004)*, pp. 494–501. IEEE Computer Society (2004)
- [50] Singh, A., Ramakrishnan, C., Smolka, S.A.: A process calculus for mobile ad hoc networks. *Science of Computer Programming* 75(6), 440 – 469 (2010), *10th International Conference on Coordination Models and Languages COORD08*
- [51] Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: *Formal Modeling: Actors, Open Systems, Biological Systems*, pp. 20–56. Springer (2011)
- [52] Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. *Fundam. Inform.* 63(4), 385–410 (2004)
- [53] Varshosaz, M., Khosravi, R.: Modeling and verification of probabilistic actor systems using pRebeca. In: *Formal Methods and Software Engineering*, pp. 135–150. Springer (2012)

- [54] Wibling, O., Parrow, J., Pears, A.: Automatized verification of ad hoc routing protocols. In: *Formal Techniques for Networked and Distributed Systems, LNCS*, vol. 3235, pp. 343–358. Springer (2004)
- [55] Wibling, O., Parrow, J., Pears, A.: Ad hoc routing protocol verification through broadcast abstraction. In: *Formal Techniques for Networked and Distributed Systems-FORTE 2005*, pp. 128–142. Springer (2005)
- [56] Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of broadcasting actors. In: *In pre-proceeding of 6th IPM International Conference on Fundamentals of Software Engineering*. pp. 114–128 (2015)